

SESSION

TOOLS AND MODELS FOR PARALLELIZATION AND INFRASTRUCTURE + POWER AWARE COMPUTING AND POWER EFFICIENCY

Chair(s)

TBA

High Performance I/O and Data Management

William W. Dai

Computer, Computational, and Statistical Sciences Division
Los Alamos National Laboratory
E-mail: dai@lanl.gov

Abstract— A library for parallel IO and data management has been developed for large-scale 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 applications, and to provide tools for the connection between applications. The high-level data structures include one- and multi-dimensional arrays, structured meshes, unstructured meshes, and the meshes generated through adaptive mesh refinement. The IO mechanism can be collective and non-collective. 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 one of MPI-IO performance.

Keywords—IO, data structure, data management, high performance.

I. 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 [1,2], SAF [3], CGNS [4,5,6], NetCDF [7,8], Silo [9], UDM [10], 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 [10]. 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.

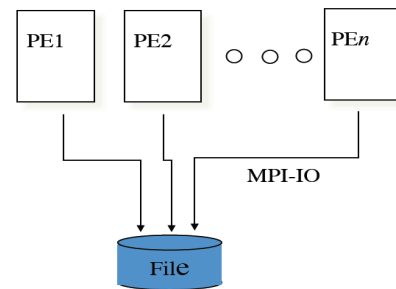


Figure 1. Data on n computer processors are written into a single file.

The HIO library writes simulation data to a single file on parallel computer environments as shown in Fig.1. It consists of functionalities for IO and data management for high-level data structures encountered in numerical simulations on parallel computer environments. 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.

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 some of our future plans.

II. FUNCTIONALITY OF THE HIO LIBRARY

The HIO library provides IO and data management tools for single and multi-dimensional arrays, structured meshes, unstructured meshes, and the mesh generated through adaptive mesh refinement (AMR) 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.

A. Functionality for Arrays

The HIO library provides IO tools for single and multi-dimensional arrays used in numerical simulations. It supports arrays with any number of dimensions. Since simulations on parallel environments often involves ghost elements, the library also supports ghost elements. Through the library, values of an array on both real elements and ghost elements may be written into data files. If instructed, only real values can be written into a file.

To get the highest possible IO performance on parallel computer environments, the library writes a multi-dimensional array on each computer processor contiguously on a disk. To keep the logical view of the multi-dimensional array for users, the library also writes the description of the array into the file, such as the size and offset in each dimension on each processor. Therefore, users get the best possible IO performance and the logical structure for each multi-dimensional array.

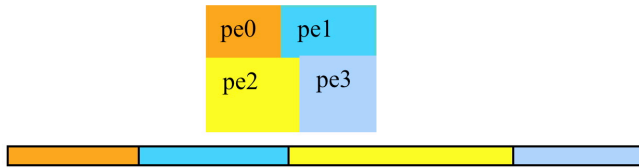


Figure 2. A 2D array distributed among 4 processors. The right is the layout of the array on disk. The data on each processor are contiguously written into the file.

To illustrate the layout of a multi-dimensional array on a disk, we take an example of a two-dimensional array distributed among four processors. As indicated in Fig.2, the data on each processor are contiguously written onto a disk file. This layout of the array on the disk lacks the global structure of the two-dimensional array. Therefore, the HIO library also stores additional information besides the array, which is called meta data. The meta data together with the data uniquely determine the global structure of the array. The meta data, together with all other possible meta data, will be written into the file when the file is closed. Therefore almost there is no additional cost to store the meta data. To avoid unnecessary meta data, all the arrays with a same processor configuration share the same meta data in a file.

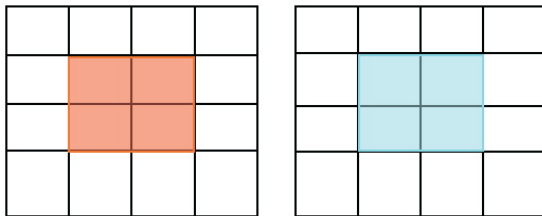


Figure 3. A 2D array distributed among 4 processors. The right is the layout of the array on disk. The data on each processor are contiguously written into the file.

For the treatment of ghost elements, Figure 3 illustrates an example, where each processor owns a 4x4 two-dimensional, but only a part of the array, 2x2, contains real data, as shown in the parts painted with colors. Other data on each processor are ghost data shown in the white. Although the ghost data are copied from real data during simulations, the ghost data are not necessarily the same as real data when the data are written into a file since ghost data may not be updated in simulations when the data are written into files. Through the library both real and ghost data may be written into a file. Furthermore, each processor may have different layers of ghost elements due to possibly different physics problems each processor is solving or different numerical algorithms used in different processors. Due to the capability to handle ghost elements, users don't have to make memory copies before writing arrays to files.

After an array is written into a file, users may query any part of data including ghost elements. For example, if a user specifies offsets and sizes in all dimensions, and if the user also requests a layer of ghost elements surrounding a part of data, the result of the query will be a part of real data surrounded by a layer of ghost data. The data on real elements are always retrieved from real elements even if this part of the array was written by more than one processor and shadowed by ghost elements.

B. Functionality for Structured Meshes

Another major part of the HIO library is the one for structured meshes and variables. In the library, writing a structured mesh and variables is very similar to writing single and multi-dimensional arrays. A structured mesh is made from possibly three one-dimensional arrays, x, y and z. Variables defined on a mesh may be represented through a one-, or two-, or three-dimensional array.

There are three aspects in which the functionality for structured meshes is different from the one for arrays in the library. First, although each processor supplies possible three arrays for x-, y- and z-coordinates to write a structured mesh, only a small number of processors actually write their data. Compared to the procedure for users to use one-dimensional array to write coordinates, this functionality makes it easier for users to write a mesh since users can treat all the processors uniformly. Second, the library doesn't write any problem-size data for a structured mesh if the mesh is uniform in all dimensions. Third, a relationship between a structured mesh and any variable associated with the mesh is automatically built in the library for future query.

The variables associated with a structured mesh include those typically used in simulation. For example, vectors defined on nodes are typically used in numerical simulations

of fluid dynamics, and vectors defined on a set of faces are often used in magneto-hydrodynamics. Variables in the HIO library may be scalars, vectors, and tensors. Each variable may be defined on zones (i.e. volumetric), or faces, or edges, or nodes, as indicated in Fig.4. The information about variables, and their associations with meshes are stored as meta data. Therefore, there is almost no cost to write the information into the file.

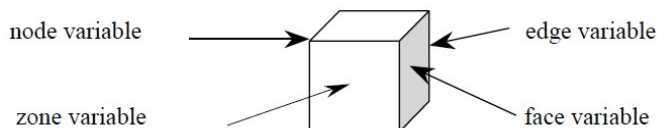


Figure 4. Illustration for variables defined on nodes, edges, faces, and zones.

The HIO library writes minimum and complete information for a structured mesh and variables defined by users. In the library, one mesh definition is used to cover all the situations for a variety of structured meshes used in applications. For example, a mesh may have ghost elements, the sizes of ghost elements on different processors do not have to be the same, and sizes of different dimensions may be different. Any dimension on any processor may be either uniform or non-uniform. The coordinates in the mesh may be either the centers of elements or grid points. The relative location of the part of a mesh on the current processor in the global structure of the mesh may be specified through either offsets or a configuration of processors in simulations.

Although the definition for structured meshes in the HIO library covers a broad range of possible meshes used in multi-physics applications, users deal with only their own definitions. All other mesh definitions are set to invalid through a function that initializes a mesh. For example, if there are no ghost elements involved in an application, users don't have to touch ghost elements at any time. If users always use coordinate arrays to indicate the grid points, users don't have to deal with any parameters for uniform meshes at any time. Any file generated through the HIO library supports more than one mesh, and each mesh may have a different set of variables.

C. Functionality for 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.

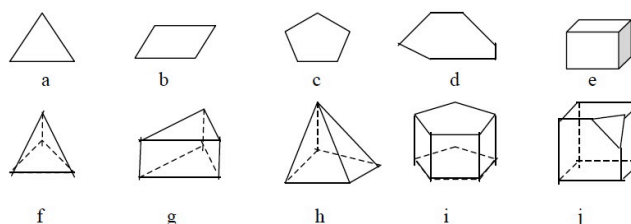


Figure 5. Examples of mesh elements supported in the HIO library for unstructured meshes.

To illustrate the meshes supported in the library, in Fig.5 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 6 shows three possible ways to make up three-dimensional elements.

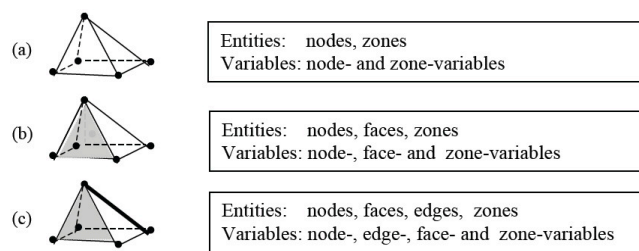


Figure 6. An element may be made from (a) nodes, or (b) faces and nodes, or (c) faces, edges, and nodes.

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.

D. Functionality for Adaptive Mesh Refinement

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

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.

E. Functionality for 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.

F. Functionality for 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.

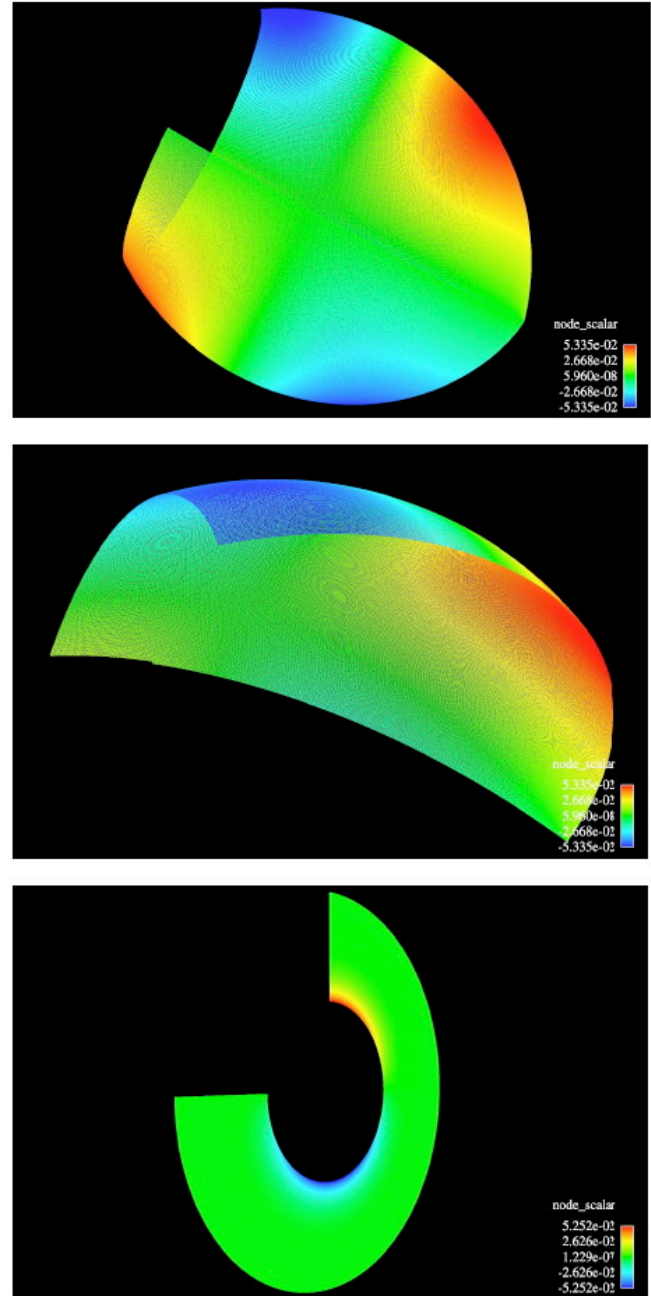


Figure 7. Three parts of an unstructured mesh with 1.6 billion elements. The left is read through an original processor rank, the middle one is read through a set of global element ids, and the right is read through a space domain.

G. Functionality for Querying

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 7 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.

H. 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.

III. BASIC FUNCTIONS AND USAGE

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_open(filename, mode, fid)`
- `hio_close(fid)`
- `hio_init(type, fileid, obj)`
- `hio_write(type, fileid, obj)`
- `hio_query(type, fileid, filter, nobjs, obj)`
- `hio_clean(type, nobjs, obj)`
- `hio_get_size(type, domain, fileid, obj)`
- `hio_read(type, domain, fileid, obj, 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.

To generate a large mesh, suppose the mesh can be generated part by part. The following segment of codes demonstrates the usage to write this mesh into a file.

```

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 file
}
hio_finalize_append(hio_umesh, m.id);

```

The segment of codes shown above in “write the part of mesh” is the following.

```

m.sizes[0] = nzone;
m.sizes[1] = nface;
m.sizes[3] = nnode;
m.num_faces_for_zone = num_faces_for_zone;
m.facelist_for_zone = facelist_for_zone;
m.num_nodes_for_face = num_nodes_for_face;
m.nodelist_for_face = nodelist_for_face;
c->coord[0] = x;
c->coord[1] = y;
c->coord[2] = z;
hio_write(hio_umesh, fileid, &m);

```

An example mesh with 1.6 billion of unstructured elements generated through 16 processors in this way is shown in Fig.8.

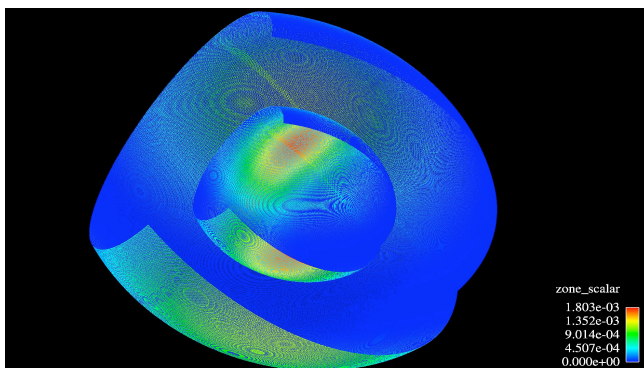


Figure 8. An unstructured mesh with 1.6 billion elements written through the “append” capability in the HIO library.

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.

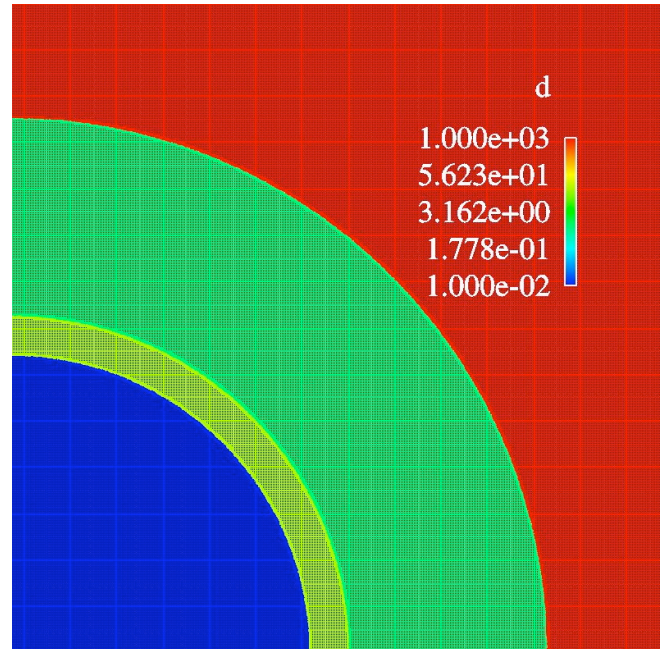


Figure 9. A cell_based AMR structured mesh with four materials and the variable of density written through the HIO library.

```

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);

```

An example mesh is shown in Fig.9 that contain four materials.

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_velocity_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);

```

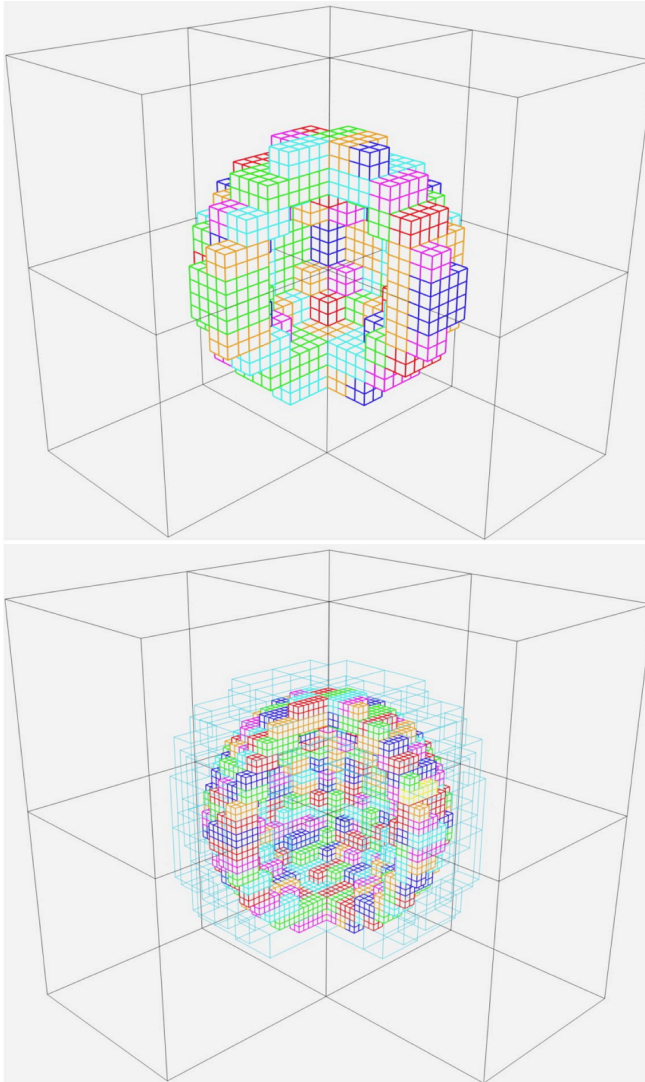


Figure 10. A patch-based AMR structured mesh written through the HIO library. The top image shows the first level of patches, and the lower one is the second level of patches on the top of the first level patches.

After a mesh and a set of variables are written into a file. 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.10. 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.10 is a patch. Figure 11 at the end of this paper shows the mesh partition of a cell based AMR structured mesh, where each rectangular with the same color contains the elements in one of 256 computer processors. The mesh is written and read through the HIO library.

IV. IO 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 the full machine, 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 supports. 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.

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 Lightning 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 resulting file size is about 405 Gbytes. As stated before, the file generated through the library is

slightly larger than the file 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, resulting file size 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 the file structure.

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 a file, the library reads all the meta data and the file structure together and reads it only once.

V. ACKNOWLEDGMENTS

The work presented here has been supported by Department of Energy through the Advanced Simulation and Computing program.

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] HDF5 home page, HDF Group, <http://www.hdfgroup.org/hdf5>
- [3] 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.
- [4] 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.
- [5] CGNS home page, <http://www.lerc.nasa.gov/www/cgns/index.html>
- [6] Thausen Th., "Parallel I/O for the CGNS System", 42nd AIAA Aerospace Sciences Meeting and Exhibit, AIAA 2004-1088, Reno, Nevada, January, 2004.
- [7] Rew, R. K., Davis, G., "NetCDF: An Interface for Scientific Data Access", *IEEE Computer Graphics and Applications*, vol.4, pp 72-82, July (1990).
- [8] NetCDF home page, <http://www.unidata.ucar.edu/software/netcdf>
- [9] Roberts, L., J., "Silo User's Guide", University of California Research Lab Report, Lawrence Livermore National Laboratory, UCRL-MA-118751-REV-1 (2000).
- [10] William W. Dai, Rob Aulwes, Michael Gaeta, and Ron Pfaff, Unified Data Model (UDM): A Library for Parallel IO and Data Management, The proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Application, Arabia et al (Eds.), CSREA Press 2007, Vol.II, pp.697-702, 2007.

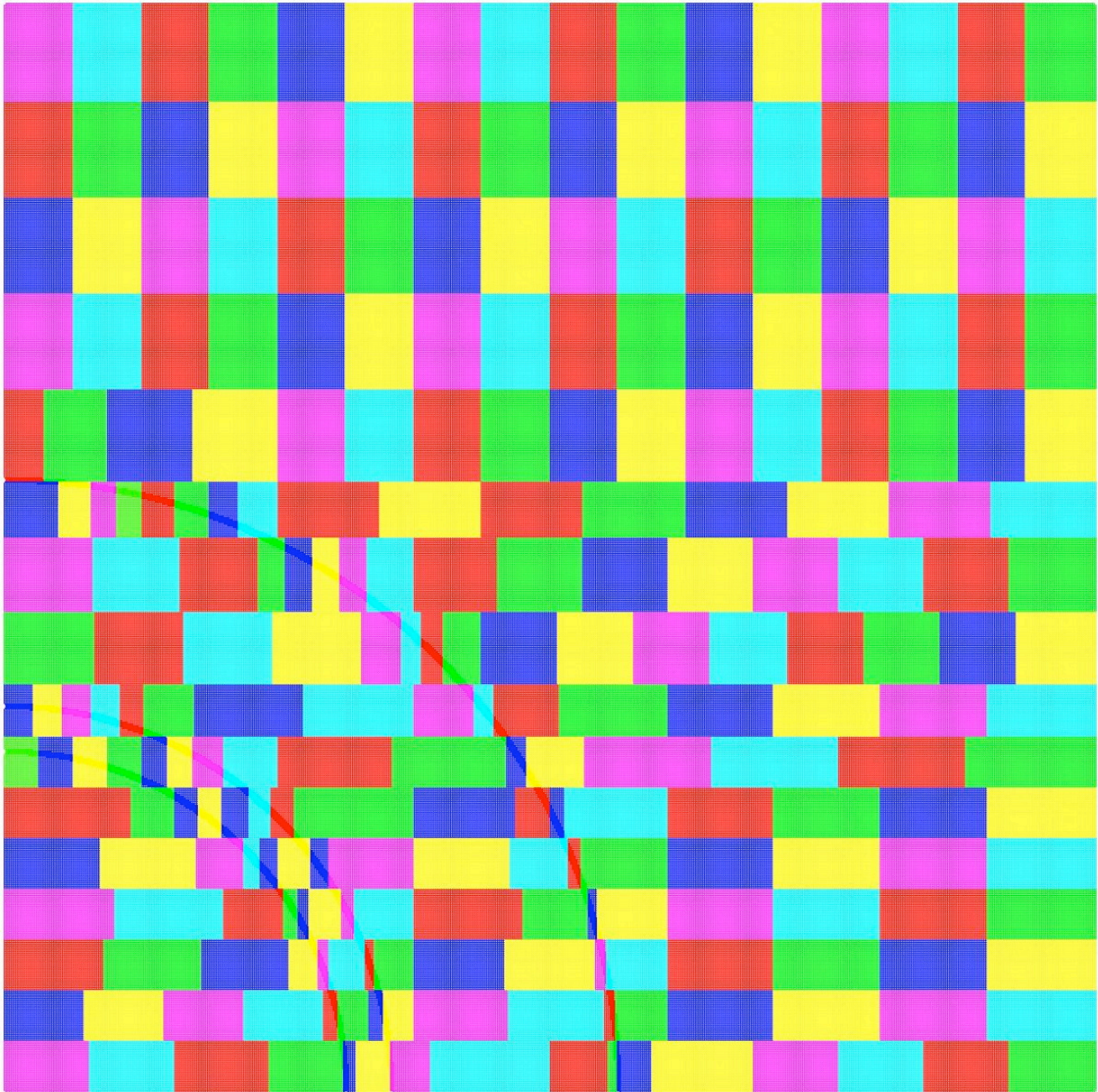


Figure 11. A cell-based AMR mesh partitioned among 256 processors. Each rectangular of a same color contains the elements in one of the 256 processors. The mesh is written and read through the HIO library.

Generation of Correct Parallel Programs Guided by Rewriting Rules

Hidekatsu Koike

Faculty of Social Information

Sapporo Gakuin University

11-banchi Bunkyoudai, Ebetsu, Hokkaido 069-8555, Japan

Email: koike@sgu.ac.jp

Kiyoshi Akama

Division of Large Scale Computing Systems

Information Initiative Center Hokkaido University

Kita 11, Nishi 5, Kita-ku Sapporo 060-0811, Japan

Email: akama@iic.hokudai.ac.jp

Abstract—In this paper, we propose a method for generation of a correct parallel program from a given specification. In this method, a sequential program is first generated by the program generation method we have developed, and then a parallel program is obtained by adding a set of abstracted parallel procedures represented by rewriting rules into the sequential program. Correctness of all the computation steps can be verified based on the equivalent transformation computation model and algorithms for correctness of parallelism. The generated parallel program can be tailored to the computation where degree of parallelism varies according to the run time situation and various procedures run in parallel with shared variables. We introduce several rewriting rules which abstract procedures for parallel computation but can be executable. The rules encapsulate complicated details of implementation taking into account computation efficiency and enable us to generate a parallel program only by adding the set of rules into an existing sequential program. Generated parallel programs are written in rewriting rules which are immediately executed by our original language system. We can even obtain imperative programs such as C++ programs by hand since the rewriting rules represent detailed procedures.

Index Terms—parallelism, programming language, equivalent transformation, formal specification, program synthesis.

PDPTA

I. INTRODUCTION

This paper introduces a method for generating a parallel program, which is correct with respect to a given formal specification, based on the equivalent transformation (ET) computation model [1] and the additional algorithms for correct parallel computation [15], [16]. In this model computation can be regarded as a sequence of ETs and a program consists of a set of ET rules and description for controls of their application. We can use variety of computational procedures with correctness theorem by using ET as a basis for computation. The ET can be seen as generalization of computation mechanism used by other computational models, for example, SLD-resolution corresponds to unfolding [2] which is an instance of ET [3]. In the ETCM, a problem to be solved is specified by a declarative description, which is a set of definite clause extended to be able to support various data structures [4], set expression and operation [5], high order expression, and logical negation [6]. A declarative description P mathematically determines its meaning $\mathcal{M}(P)$, which is intuitively an extension of the declarative semantics of

logic programs. The formal definition can be found in several papers related to the ETCM, for example [1], [3]. A rewriting rule is an ET rule with respect to a declarative description P iff the rule rewrites P into P' and $\mathcal{M}(P) = \mathcal{M}(P')$. We can use various ET rules for computation to improve program efficiency since ET rules can represent more detailed procedures than by clauses. We let our programming language have the rule syntax to represent a variety of ET rules and run efficiently by adopting specified head pattern, one-side matching, and applicability conditions. A program is always correct if it consists of all ET rules since correctness of a rule is completely independent from each other. The independency enables us to check correctness of an ET rule, which is newly generated, and then add into a whole program one by one. In addition, the independency allows us freely combine ET rules that seem to run the fastest for a given problem. Given a specification as a declarative description, a hand written set of rewriting rules may be correct if carefully constructed to preserve the semantic meaning of the specification. We can check correctness of each rule with rigorous theory [3], [7], [8], [9] and even can generate a wide class of ET rules including non-trivial ones automatically [10].

Computation in the ETCM is generally nondeterministic since we have to determine which rule is to be applied to which atom in which clause. Such non-determinism can be processed in parallel. Selections of clause and atom provide the sources of OR-parallelism and AND-parallelism, respectively which bear a close resemblance to those in parallel logic programming [11], [12], [13], [14]. In this paper, we propose a method for generation of a correct parallel program from a given specification. In this method, a sequential program is first generated by the program generation method we have developed [10], and then a parallel program is obtained by adding a set of abstracted parallel procedures represented by rewriting rules into the sequential program. Correctness of all the computation steps for parallelism can be verified if each of the rewriting rules follows the theories proposed in [15], [16]. The generated parallel program can execute the computation where degree of parallelism varies according to the run time situation and various procedures run in parallel with shared variables. We use nonogram puzzles as an example of such situations. We introduce several rewriting rules which abstract

procedures for parallel computation. The rules encapsulate complicated details of implementation taking into account computation efficiency and enable us to generate a parallel program by adding the set of the rules into an existing sequential program. The introduction of the rules gives a simplified framework for constructing a parallel program from a sequential program based on the ETCM. The method is especially useful in situation where we need both rigorously correct results and adequate efficiency.

II. PROBLEM SOLVING BY ETCM

A. Definition of Objects

We here define the syntax of objects appearing in specifications and rules by the following pseudo-EBNF. A language interpreter system based on the ETCM named ETI [17] accepts objects and rules which are written in the following syntax and ones introduced later.

```

<alphabet> ::= [a-zA-Z]
<number> ::= [0-9]
<symbol> ::= [+*~+= ...]
<name> ::= (<alphabet> | <number> | <symbol>)+
<var> ::= "*" <name>? | "?"
<term> ::=
  (<name> | <number>+ | <var>)+ <term>*
<atom> ::= "(" <name> <term>* ")"
<clause> ::= <atom> "<-->" <atom>* "."

```

In the pseudo-EBNF, we apply $*$, $+$, $?$ operators which mean zero or more, one or more, and optional respectively, and other regular-expression-like operators. $\langle \text{name} \rangle$ is a name which is used to represent a name of variable or atom. It also can be an operative symbol such as $=$, $-$, $+$, etc. $\langle \text{var} \rangle$ is a variable which has an asterisk as a prefix of its name. A variable is anonymous if a variable has $?$ as its name. $\langle \text{term} \rangle$ is a term appearing in an argument of an atom. $\langle \text{atom} \rangle$ is an atomic formula. In this paper, an atomic formula is referred to as an atom. $\langle \text{clause} \rangle$ is a definite clause which has a head as an atom and a body as a possibly empty set of atoms. The head and body of a definite clause are separated by $\langle \text{--} \rangle$.

B. Substitution

Given substitutions θ_1 and θ_2 , $\theta_1 \circ \theta_2$ denotes the composition of θ_1 and θ_2 . $t\theta$, $a\theta$, and $cl\theta$ denote substitutions of term t , an atom a , and a clause cl by using a substitution θ , respectively.

C. Nonogram

Nonogram is a puzzle to reveal a picture in an $n \times m$ grid implied by clue sequences of integers given at the side of the grid. Fig. 1 (a) shows an example of nonogram and Fig. 1 (b) shows the answer to (a) in which \blacksquare represents a filled cell and \times represents a blank cell. Nonogram can be represented by sequences of variables corresponding to rows and columns in a grid and clue sequences of non-negative integers which are given on each row and column. Answering a nonogram is to determine each variable value according to the given numbers. Each value represents fill or blank. Each number specifies the number of cells which must be connected. Each group of cells must be separated by one or more blank cells.

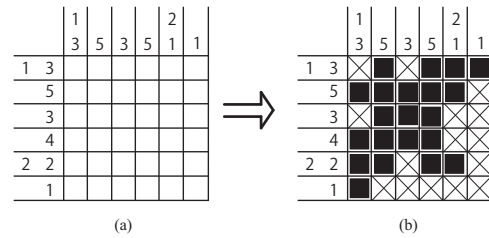


Fig. 1. Example of Nonogram

We use a program for solving nonogram as a sample which has AND-parallelism with synchronization overheads. The degree of parallelism in solving a nonogram puzzle can vary since threads share variables with each other and a certain thread might need specialization of variables made by other threads for further computation; this possibility depends on a given specific puzzle problem and computational situation. The application of the proposed approach is not limited to the puzzle.

D. Formal Specification as a Declarative Description

In the ETCM, a declarative description is neither a program nor has procedural semantics; thus, orders of atoms and clauses are not significant, and a declarative description only determines the meaning of a given specification mathematically. In contrast, a set of rules is a program which rewrites the declarative description preserving its meaning. A simple example of declarative description $P = D \cup Q$ is as follows.

```

D = {
(pat () *pl)<--(allZero *pl).
(pat (*n | *ns) (0 | *pls))
  <--(check_c *ns *pl),(pat (*n | *ns) *pls).
(pat (*n|*ns) (1 | *pls))
  <--(sub *n 1 *n2),(seq1 *n2 *pls *rest),
  (start0 *rest), (pat *ns *rest).
(seq1 *n (1|*pls) *pls2)
  <--(> *n 0), (:= *n2 (- *n 1)),
  (seq1 *n2 *pls, *pls2).
(seq1 0 *pls *pls2)<--(= *pls *pls2).
(start0 ())<--.
(start0 (0|*rest))<--.
(allZero ())<--.
(allZero (0 | *rest))<--(allZero *rest).
(check_c *ns *pl)<--
  (len *ns *n1), (len *pl *pl1),
  (listSum *ns *sum), (sub *n1 1 *n12),
  (add *sum *n12 *m1), (>= *pl1 *m1).
}
Q = {
(ans (*11 *12 *13 *14 *15 *16
  *21 *22 *23 *24 *25 *26
  *31 *32 *33 *34 *35 *36
  *41 *42 *43 *44 *45 *46
  *51 *52 *53 *54 *55 *56
  *61 *62 *63 *64 *65 *66))<--
(pat (1 3) (*11 *12 *13 *14 *15 *16)),
(pat (5) (*21 *22 *23 *24 *25 *26)),
(pat (3) (*31 *32 *33 *34 *35 *36)),
(pat (4) (*41 *42 *43 *44 *45 *46)),

```

```

(pat (2 2) (*51 *52 *53 *54 *55 *56)),
(pat (1) (*61 *62 *63 *64 *65 *66)),
(pat (1 3) (*11 *21 *31 *41 *51 *61)),
(pat (5) (*12 *22 *32 *42 *52 *62)),
(pat (3) (*13 *23 *33 *43 *53 *63)),
(pat (5) (*14 *24 *34 *44 *54 *64)),
(pat (2 1) (*15 *25 *35 *45 *55 *65)),
(pat (1) (*16 *26 *36 *46 *56 *66)).
}

```

P consists of the sets of clauses D and Q . D declaratively defines the rule of Nonogram. A filled cell ■ and blank cell × are represented by 1 and 0, respectively. D is never changed in computation. Q presents a specific nonogram puzzle to be solved and is changed by rewriting rules at run-time. P implicitly determines the answer to Q . The answer is obtained by repeatedly rewriting Q until the answer is obvious, e.g. Q is transformed into Q' by more than one rewriting as follows.

```

Q' = {
(ans (0 1 0 1 1 1
      1 1 1 1 1 0
      0 1 1 1 0 0
      1 1 1 1 0 0
      1 1 0 1 1 0
      1 0 0 0 0 0))<- .
}

```

E. ET Rules

In the ETCM, a program is a set of ET rules while a declarative description only determines the meaning. An application of an ET rule rewrites a set of clauses while substitutions are often generated at the same time. We use two sorts of ET rules, D (deterministic) rules and N (non-deterministic) rules for both efficiency and simplicity. The D-rules represent primitive procedures (including substitutions, arithmetic operations, I/O access, and so on.) and user-defined procedures in a deterministic way, and can be invoked by N-rules which can specify procedures in a non-deterministic manner and yield a result set of more than one element as opposed to D-rules whose number of results is always one. The application of D-rules is strictly deterministic. The computation by the D-rules transforms the leftmost atom in the body of a clause. If there are two or more applicable D-rules, the topmost rule is selected. The atom transformed by D-rules is called a D-atom. A D-rule, which is built into the system, is called a built-in D-rule or B-rule. A D-rule has the following syntax.

```

<D-rule> ::= <d_head><d_cond>
           "-->" <d_body> "."
<d_head> ::= <D-atom>
<d_cond> ::= <empty> | ", {" <D-atom>+ "}"
<d_body> ::= <empty> | <D-atom> |
           <D-atom> ", " <d_body>
<D-atom> ::= <atom>

```

Note that the definition of $\langle \text{atom} \rangle$ is inherited from the above definition. An N-rule is non-deterministic. Thus the applicability condition depends on the current situation of the computation. N-rules rewrite a set of clauses and have

the ability to create one or more clauses from one clause to obtain all the answers. This can be seen as a guarantee of completeness from the point of view of logic programming. N-rules can call a sub-computation done by the D-rules to check their applicability condition and to execute their more specified application. An atom in a clause, which is transformed by N-rules, is called an N-atom. The result of the sub-computation can propagate through variable substitutions. An N-rule has the following syntax.

```

<N-rule> ::= <name>?<head><cond><bodies>
<head>   ::= <N-atom>("," <N-atom>)*
<cond>   ::= ("{" <D-atom>+ "}")?
<body>   ::= (<exec>("," <rep>))/?
<rep>    ::= <N-atom>*
<bodies> ::= "==" <body> "." |
           "==" <body> ";" <bodies>
<exec>   ::= ("{" <D-atom>+ "}")?
<N-atom> ::= <atom>

```

Note that the $\langle \text{atom} \rangle$, $\langle \text{D-atom} \rangle$, and $\langle \text{name} \rangle$ are inherited from the above definitions.

F. Application of D-Rules

We here explain how the rules are applied during computation. Let T be the set $\{true, false\}$ and $exec$ a mapping from a sequence of atoms to $S \times T$ where S is the set of all the substitutions. The mapping $exec$ is recursively defined later. Let sub be a mapping from $S \times T$ to S and val a mapping from $S \times T$ to T . The mappings sub and val are defined as follows.

Given $(s, t) \in S \times T$, $sub((s, t)) = s$ and $val((s, t)) = t$.

Let sa be a sequence of D-atoms such as $sa = a_1, \dots, a_n$, where $n \geq 0$, dcl be a clause of the form $dh \leftarrow sa$ where dh is a D-atom. A D-rule is applicable iff it satisfies all the following conditions:

- There exists the most general substitution $\theta_h \in S$ such that $\langle \text{d_head} \rangle \theta_h = a_1$.
- $val(exec(\langle \text{d_cond} \rangle \theta_h)) = true$.
- The rule appears first among the rules satisfying the above conditions in the source code.

The application procedure of a D-rule is as follows. Let θ_c be a composite substitution obtained by the above applicability check such as

$$\theta_c = sub(exec(\langle \text{d_cond} \rangle \theta_h)) \circ \theta_h.$$

Let $\langle \text{d_body} \rangle$ be a sequence of atoms such as $\langle \text{d_body} \rangle = b_1, \dots, b_m$ where $m \geq 0$. A new clause dcl' is obtained from dcl by the D-rule as follows.

$$dcl' = (dh \leftarrow b_1, \dots, b_m, a_2, \dots, a_n) \theta_c.$$

Primitive data operations from explicit substitution and mathematical operations to file I/O are realized by B-rule. A B-rule is a special D-rule having an empty body and yields (possibly empty) substitution according to the arguments in a D-atom to be rewritten. We obtain dcl' from dcl by a B-rule as follows.

$$dcl' = (dh \leftarrow a_2, \dots, a_n)\theta_b$$

where θ_b is substitution which is generated by application of the B-rule and defined in its respective rule. The θ_b may vary according to the arguments in the target D-atom which is rewritten by the B-rule if the atom has one or more arguments.

The mapping *exec* is recursively defined as follows. Assume that no D-rules or B-rules are applicable after more than one attempt to apply a rule to a clause *dcl*. Let θ_e be a composite substitution of all the substitutions obtained by the one or more attempts of applications of rules.

- If *sa* is empty, then $exec(sa) = (\theta_e, true)$.
- Otherwise $exec(sa) = (\theta_e, false)$.

Thus B-rules and D-rules define the mapping *exec* recursively.

G. Application of N-Rules

Let *C* be a clause of the form $a \leftarrow \{b\} \cup Bs$ where *a* is the head atom of *C*, *b* is a body atom of *C*, and *Bs* is a set of body atoms of *C*, and $Q = \{C\} \cup Cs$ where *Cs* is a set of clauses. Assume that *exec* is given. An N-rule is applicable iff it satisfies all the following conditions:

- There exists the most general substitution θ_h such that $\langle head \rangle \theta_h = b$.
- $val(exec(\langle cond \rangle \theta_h)) = true$.
- The rule has the highest priority of the rules satisfying the above conditions.

Note that the applicability check never affects the target clause. The application procedure of an N-rule is as follows. Let $\theta_c = sub(exec(\langle cond \rangle \theta_h)) \circ \theta_h$. Assume that the rule has *n* bodies. For each $i \in \{1, \dots, n\}$, if $val(exec(\langle exec_i \rangle \theta_c)) = true$, then a new clause $C_i = (a \leftarrow (\langle rep_i \rangle \theta_c) \cup Bs)\theta_{e_i}$ is created, where $\theta_{e_i} = sub(exec(\langle exec_i \rangle \theta_c)) \circ \theta_c$.

Then finally, *C* is replaced with the set of the new clauses. If no new clauses are created, then *C* is merely removed. Thus,

$$Q' = \{C_i \mid (i \in \{1, \dots, n\}) \& \\ val(exec(\langle exec_i \rangle \theta_c)) = true\} \cup Cs$$

is obtained from *Q* by application of the rule.

This replacement will be repeated until no rule is applicable or the clauses run out. Note that the applicability of a rule is specified by a combination of head matching and the execution of the applicability condition part, enabling very detailed control of computational flow. The atoms in the same clause can share common variables, therefore when a rule is applied to an atom and variable substitution occurs, it affects the other atoms and often makes it possible to apply other rules to them. With this interaction, flexible computation is achieved.

H. Rule Priority

Applicability control of an N-rule can be specified in detail by its head atom and applicability condition part. However there is a situation where a program is simplified as a whole if a way to specify control more globally exists; therefore the way to specify priority of rule is introduced. Specification of rule priority consists of the following steps:

- 1) Priority groups are made and each group is given a priority order.

- 2) Each rule to be given a priority order is given a name.
- 3) Each rule is assigned its priority group by using its name.

The first step is realized by RuleClassOrder directive which has variable arguments. The third step is realized by RuleClass/2. An example of them is shown in Section III. In that case, these directives make two priority groups named 0 and 1, and then define that default rule priority group is 0 and the rule named pat2 is assigned into group 1 which means the rule is lowest priority among all the other rules.

III. AN EXAMPLE OF ET RULES

We have developed a rule generation method [10] by which we can obtain ET rules with respect to a given formal specification described in declarative description. The following set of rewriting rules is generated by the method.

```
?-(RuleClassOrder 0 1),(RuleClass pat2 1),
(RuleClass otherwise 0).

(pat (*n|*ns) ())==>{(false)}.
(pat () *PL)==>(allZero *PL).

pat2
(pat (*n|*ns) (*p|*pls))
==>{(= *p 0)},(check_const (*n|*ns) *pls),
(pat (*n|*ns) *pls);
==>{(= *p 1),(:= *n2 (- *n 1))},
(seq1 *n2 *pls *rest),(start0 *rest),
(pat *ns *rest).

check_const
(check_const *n *pl),
{(length *n *n1),(length *pl *p11),
(listSum *n *ns),(:= *n12 (- *n1 1)),
(:= *ns2 (+ *ns *n12))}
==>{(>= *p11 *ns2)}.

(seq1 *N *PL *PL2),{(> *N 0)}
==>{(= *PL (1 | *PLS)),(:= *N2 (- *N 1))},
(seq1 *N2 *PLS *PL2).
(seq1 0 *PL *PL2)==>{(= *PL *PL2)}.

(start0 ())==>.
(start0 (*p | *rest))==>{(= *p 0)}.

(allZero ())==>.
(allZero (*p | *rest))==>{(= *p 0)},
(allZero *rest).

(length () *1)-->(= *1 0).
(length (*a | *rest) *1)-->
(length *rest *11),(:= *1 (+ *11 1)).

(listSum () *s)-->(= *s 0).
(listSum (*n | *rest) *s)-->
(listSum *rest *s1),(:= *s (+ *n *s1)).
```

The first line which starts “?” is a directive set for prioritizing rules. This is obtained by following the policy that as a rule has more bodies, as it has lower priority. The reason why we adopt such policy is that a rule having the plural number of bodies increases computation complexity than a single body rule; thus efficient computation can be done by the prioritization of the rules.

IV. PARALLELISM OF COMPUTATION IN ETCM

Computation in the ETCM has three kinds of non-determinism: which clause, which atom, and which rule are selected. In this paper, we treat non-determinism arising from atom selection. This kind of non-determinism can be transformed into AND-parallelism. In the ETCM, non-determinism from clause selection can be also transformed into AND-parallelism since clause selection can be reduced to atom selection of which scope ranges over all the atoms in all the bodies of a whole set of clauses.

V. OUTLINE OF PARALLELIZATION

To obtain a parallel program from a formal specification, we prepare additional information in addition to the specification. The information consists of:

- The maximum number of processes can be run in parallel,
- The atom to be processed in parallel, and
- Specification of the form of result which a server process sends to a client.

The second and third information are given as a set of template rules for parallelism which are described in Section VI.

VI. ABSTRACTION OF PARALLEL COMPUTATION BY REWRITING RULES

A. Communication Object

A communication object is used to transfer data between a client process and a server process. The object encapsulates the procedures for parallel communication between a client and a server such as mutual exclusion. The object consists of a pointer to a server, a container for a result returned by a server, and a flag showing the termination of a server.

B. Requesting Rules

Applicability condition for atoms to be processed in parallel and procedures representing how to process the atoms are generalized. Applicability conditions are as follows:

- An atom which has not been rewritten yet is always applicable.
- An atom which has already rewritten is applicable iff its shared variable is substituted by processing other atoms.

The first condition and respective procedure are represented by the following rule:

```
(pat *n *cs)==>(pat *n ? *cs).
```

The rule rewrites two-argument-`pat` atom into three-argument-`pat` atom whose second argument represents the previous state of cells while the third argument represents the current state of the cells. The new atom is applicable since the third argument is more specialized than the second argument which means the atom has new state and is need to check whether new substitution is obtained from the atom. The second condition and respective procedure are represented by the following rule:

```
(pat *n *pcs *cs),
{(specialized *pcs *cs),
 (sendReq *co (pat *n *cs))}
==>{(copy *cs *npcs)},(pat *n *npcs *cs *co).
```

The rule is applicable to three-argument-`pat` atom iff the third argument is more specialized than the second argument and a request of processing two-argument-`pat` atom is successfully sent to a server. When the rule is applied, it copies the current state of cells represented by the variable `*cs` to `*npcs` and replace the original atom with a four-argument-`pat` atom which has `*npcs` and `*co` which is an object for communication with a server in addition to the original arguments. The B-rule `specialized/2` checks whether the second argument is more specialized than the first argument. The B-rule `sendReq/2` encapsulates a procedure to send a request to a server for a parallel computation. The procedure is defined as follows.

- 1) Perform none-blocking P operation.
- 2) If the P operation failed then return *false* otherwise go to the next step.
- 3) Get a server process id.
- 4) Create a communication object associated to the server process id.
- 5) Set a termination flag in the communication object to be *false*.
- 6) Make a copy of the second argument and send the copy to a server.
- 7) Unify the first argument and the communication object.
- 8) Return *true*.

C. Receiving Rules

In this section, receiving a computation result from a server is abstracted by the following rule:

```
(pat *n *pcs *cs *co),
{(terminated *co)}
==>{(copy *cs *cs2), (getResult *co *cs),
 (makePCS *pcs *cs2 *cs *npcs)},
(pat *n *npcs *cs).
```

The rule is applicable to a four-argument-`pat` atom iff computation of the server represented by the communication object `*co` is terminated. When applied, the rule copies the current state `*cs` to `*cs2`, obtains a computation result from server and unifies the result and `*cs`, makes a previous state of cells as a variable `*npcs` according to `*pcs`, `*cs`, and `*cs2`, and replaces the original atom with a new three-argument-`pat` atom. The B-rule `terminated/1` checks whether computation of the server specified by its argument is terminated. The B-rule `getResult/2` obtains the result from the server and unifies the result and the second argument. The B-rule `makePCS/4` makes a new previous state to be the second argument of the new three-argument `pat` atom according to substitution made from the outside of the atom while a server process runs and that from the server. If either the outside substitution or the server substitution occurs, then the new atom is applicable unconditionally by letting the new previous state `*npcs` be a variable; as a result the current state is always more specialized than the previous state.

D. Server Rules

A server processing a requested computation from a client runs in parallel in a process. The process means

a conceptual parallel process and can be replaced with a thread at implemental stage for example. A server receives an atom via `sendReq/2` called by a client, and then makes a clause from the atom. From example, if `(pat (1 3) (*1 *2 *3 *4 *5 *6))` is received, the following set Q_s of a clause is made:

```
Qs = {
(pat (1 3) (*1 *2 *3 *4 *5 *6))
  <--(pat (1 3) (*1 *2 *3 *4 *5 *6)).
}
```

By repeated application of ET rules in Section III to Q_s , Q_s' is obtained as follows.

```
Qs' = {
(pat (1 3) (1 0 1 1 1 0))<--.
(pat (1 3) (1 0 0 1 1 1))<--.
(pat (1 3) (0 1 0 1 1 1))<--.
}
```

Q_s' represents three answers. The server sends a result made from Q_s' . A way how to make a result from a set of clauses depends on the characteristics of the problem to be solved, so that the way must be appropriately specified according to the problem. In this case, a result is made by determining common values. For example, result `(? ? ? 1 1 ?)` is made from Q_s' since the fourth and fifth elements of the second arguments in the `pat/2` atoms are all 1. The result is sent via a communication object, i.e. the result is set to the container in a communication object and then the termination flag is set to be *true* to notify a client of a termination of the server process.

VII. EXECUTION OF A PROGRAM

The above mentioned set of ET rules represents procedures for solving a nonogram puzzle in parallel, so that we can regard the set as a parallel program. The program can be executed in the following ways.

A. ETI

We have developed an interpreter system ETI [17] which executes a set of ET rules directly. In this way, a completely correct parallel program can be obtained with respect to a given specification if the above mentioned rules for parallelism are proved their correctness. The correctness can be verified by checking each rule following the procedure described in [18], [16]. Note that the number of the rules for parallelism is limited in a whole program and the other rules can be generated by the program generation method [10]; thus verification cost is reasonably low and we can obtain a program which is correct with respect to a given specification at low cost.

B. Generating an Imperative Program

When efficiency is the most important factor, the advantage of the proposed framework still exists. A set of rules represents a parallel procedure in detail. This is a very detailed guide to writing a correct parallel program when human translate a set of rules into an imperative program such as C++ and Java programs.

Processor	Intel Xeon X5355 2.6GHz 8 cores (2 Processors)
Physical memory	8GB
OS	Windows Server 2008 R2 Enterprise 64bit
Compiler	Visual Studio 2010 SP1 beta 64bit
Multithreading Lib	boost 1.45

TABLE I
EXPERIMENTAL ENVIRONMENT

Q No.	size	1	2	4	8
23179	30×30	101641	82389	54131	48207
22908	30×30	57239	33131	21891	14132
22528	25×25	8070	5409	4864	4774
22560	25×25	24193	25129	27473	33299
22551	25×25	38299	24960	25558	41131
13960	25×25	53710	36860	27036	26351
23329	30×30	30606	15940	9242	8792

TABLE II
EXPERIMENTAL RESULTS

When translates into C++, for example, classes representing a term, an atom, a clause, and a set of clauses are prepared first. After that, the methods corresponding to each rule are written in the appropriate places of a program which might be methods of each class.

VIII. EXPERIMENTAL RESULTS

We have developed a program written in C++ transferred from the above-mentioned set of rules and run the program in an 8-core processor computer. Table I shows the specification of the experimental environment. The program finds all the answers to a given puzzle. In other words, the program guarantees that another answer does not exist besides found answers.

We used nonogram puzzles published in the web site at <http://www.minicgi.net/logic/>. Table II shows execution times the program takes for each puzzle. The first column shows question numbers—we can obtain the URL of the original source of the corresponding puzzle by connecting “<http://www.minicgi.net/logic/logc/>”, a number in the column, and “.html”. The second column shows the size of puzzle. The third to sixth columns show the execution times (in msec) took to solve each puzzle by using the number of threads corresponding to the number on the top of column. The execution times of question number 22560 are not improved by any parallel computation. The reason is that the problem can be solved deterministic way and essentially not for parallel computation. Similarly the reason why these execution times are not improved as much as the number of threads is that there is determinism in processing a nonogram puzzle in some degree where no parallel processes are applied. The proposed method can treat this kind of problem where the degree of parallelism varies at run time using as may resources as possible.

IX. COMPARISON

In the many parallel logic programming languages [12], [11], [19], [14], the executable codes for parallel processing are to be written by programmers and verifying their correctness is often far more complicated than that of sequential codes since they are not included in first-order logic that is the base of correctness of LP. On the other hand, in our method, every computation step can be verified by the rigorous theory and efficiency of computation can be obtained by ET rules which can specify detailed control with a consistent theory.

X. CONCLUSION

In this paper, we proposed a method for generate a correct parallel program from a given formal specification. In this method, a parallel program is made by adding abstracted parallel procedures, which are represented by rewriting rules, into a sequential program which is a set of ET rules. A program which is a set of ET rules means that its correctness is guaranteed. The correctness of the abstracted parallel procedure is verified by following the theories and algorithms described in [15], [16]. This paper also shows a set of executable rewriting rules for parallel computation which follows the theories and algorithms and encapsulates details of implementation such as the use of specific library and directives for parallelism. Our future work is to generate an efficient imperative program (such as C++) from a set of rewriting rules automatically.

ACKNOWLEDGMENT

The work is partly supported by the collaborative research program 2010, information initiative center, Hokkaido University, Sapporo, Japan.

REFERENCES

- [1] K. Akama, T. Simizu, and E. Miyamoto, "Solving problems by equivalent transformation of declarative programs," *Journal of the Japanese Society for Artificial Intelligence*, vol. 13, pp. 944–952, 1998.
- [2] A. Pettorossi and M. Proietti, "A theory of logic program specialization and generalization for dealing with input data properties," in *Selected Papers from the International Seminar on Partial Evaluation*. Springer-Verlag, 1996, pp. 386–408.
- [3] K. Akama, E. Nantajeewarawat, and H. Koike, "A class of rewriting rules and reverse transformation for rule-based equivalent transformation," *Electronic Notes in Theoretical Computer Science*, vol. 59(4), pp. 1–16, 2001.
- [4] K. Akama, H. Koike, and H. Mabuchi, "Equivalent transformation by safe extension of data structures," *Perspectives of System Informatics, Lecture Notes in Computer Science*, vol. 2244, pp. 140–148, 2001.
- [5] H. Koike, K. Akama, and H. Mabuchi, "Multi-computation mechanism for set expressions," in *International Conference on Computing and Information Technologies (ICCIT 2001)*, 2001, pp. 391–397.
- [6] K. Hidekatsu, A. Kiyoshi, M. Hiroshi, O. Koichi, and S. Yoshinori, "A theoretical foundation of problem solving by equivalent transformation of negative constraints," *Transactions of the Japanese Society for Artificial Intelligence*, vol. 17, pp. 354–362, 20021101. [Online]. Available: <http://ci.nii.ac.jp/naid/10015771293/en/>
- [7] K. Akama, E. Nantajeewarawat, and H. Koike, "Componentwise program construction: Requirements and solutions," in *WSEAS Transactions on Information Science and Applications*, ser. Issue 7, vol. 3, 2006, pp. 1214–1221.
- [8] —, "Program generation in the equivalent transformation computation model using the squeeze method," in *PSI 2006*, ser. Lecture Notes in Computer Science, vol. 4378. Springer-Verlag, 2007, pp. 41–54.
- [9] K. Akama and E. Nantajeewarawat, "Formalization of the equivalent transformation computation model," *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 10, no. 3, pp. 245–259, 2006.
- [10] H. Koike, K. Akama, and E. Boyd, "Program synthesis by generating equivalent transformation rules," in *Proceedings of the Second International Conference on Intelligent Technologies (InTech'01)*, 2001, pp. 250–259.
- [11] F. Bueno, M. G. de la Banda, and M. Hermenegildo, "Effectiveness of abstract interpretation in automatic parallelization: a case study in logic programming," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 2, pp. 189–239, Mar. 1999. [Online]. Available: <http://www.acm.org/pubs/citations/journals/toplas/1999-21-2/p189-bueno/>
- [12] J. C. de Kergommeaux and P. Codognot, "Parallel logic programming systems," *ACM Comput. Surv.*, vol. 26, pp. 295–336, September 1994. [Online]. Available: <http://doi.acm.org/10.1145/185403.185453>
- [13] G. Gupta, E. Pontelli, K. A. Ali, M. Carlsson, and M. V. Hermenegildo, "Parallel execution of prolog programs: a survey," *ACM Trans. Program. Lang. Syst.*, vol. 23, pp. 472–602, July 2001. [Online]. Available: <http://doi.acm.org/10.1145/504083.504085>
- [14] B. Ramkumar and L. V. Kalé, "Machine independent and and or parallel execution of logic programs: Part ii-compiled execution," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, pp. 181–192, February 1994. [Online]. Available: <http://portal.acm.org/citation.cfm?id=628917.629259>
- [15] K. Akama, E. Nantajeewarawat, and H. Ogasawara, "Generation of correct parallel programs based on specializer generation transformations," in *Proceedings of the 7th international conference on intelligent technologies (InTech'06)*, 2006, pp. 90–99.
- [16] H. Ogasawara, K. Akama, and H. Mabuchi, "Specialization-based parallel processing without memo-trees," *International Journal of Electrical and Computer Engineering*, vol. 4, no. 8, pp. 518–523, 2009.
- [17] H. Koike, K. Akama, and H. Mabuchi, "A programming language interpreter system based on equivalent transformation," in *2005 IEEE 9th International Conference on Intelligent Engineering Systems (INES 2005)*, 2005, pp. 283–288.
- [18] K. Akama, E. Nantajeewarawat, and H. Ogasawara, "Generation of correct parallel programs based on specializer generation transformations," in *Proceedings of the 7th international conference on intelligent technologies*, 2006, pp. 90–99.
- [19] G. Gupta, K. A. M. Ali, M. Carlsson, and M. V. Hermenegildo, "Parallel execution of prolog programs: A survey," *ACM Transactions on Programming Languages and Systems*, vol. 23, p. 2001, 1995.

LCPI values allow PerfExpert not only to determine which code sections suffer from performance bottlenecks but also to narrow down the cause of the poor performance to specific categories such as data accesses or branch instructions. Figure 1 shows how PerfExpert presents the results of its analysis of a triply-nested loop that performs a matrix-matrix multiplication. Longer bars represent higher fractions of runtime spent in executing the corresponding class of operations. Figure 2 lists the source code of the assessed loop nest. For illustration purposes, we used a poor loop order and no optimizations so that executing the code will result in bad memory access patterns. Indeed, PerfExpert detects these weaknesses and correctly identifies data accesses and TLB accesses as the primary culprits.

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[i][j] += a[i][k] * b[k][j];
```

Figure 2: Simple triply-nested loop for matrix multiplication

There is, however, often a substantial gap between identification of a problem and its resolution. For example, how should one resolve the problem of poor performance due to the TLB? Many application programmers do not know, nor should they have to know, what exactly a TLB is. Which source-code statements can cause data TLB problems and how can they be rewritten to yield better performance? The difficulty of such questions is compounded when multiple categories are reported to be a problem at the same time.

Loop in function main() at mmm.c:25 (100% of total runtime)
change the order of loops loop i { loop j {...} } → loop j { loop i {...} }
employ loop blocking and interchange loop i { loop k { loop j { c[i][j] = c[i][j] + a[i][k] * b[k][j]; } } → loop k step s { loop j step s { loop i { for (kk = k; kk < k + s; kk++) { for (jj = j; jj < j + s; jj++) { c[i][jj] = c[i][jj] + a[i][kk] * b[kk][jj]; } } } } componentize loops by factoring them into their own subroutines ... loop i { ... } ... loop j { ... } ... void li() { ... }; void lj() { ... }; ... li(); ... lj(); ...
apply loop fission so every loop accesses just two different arrays loop i { a[i] = a[i] * b[i] - c[i]; } → loop i { a[i] = a[i] * b[i]; } loop i { a[i] = a[i] - c[i]; }

Figure 3: Abridged AutoSCOPE output with code examples

To make it easier and quicker to resolve such performance problems, we designed AutoSCOPE, which is accessible through a simple web interface [21]. It analyzes the output of PerfExpert and determines which categories are in need of optimization. Then, it retrieves and ranks relevant suggestions for those categories from an annotated optimization database and selects the most relevant recommendations based on their ranks. The final set of suggestions is presented to the user. For example, AutoSCOPE “knows” that the probable cause of the TLB access bottleneck is a very long data access stride to an array. Thus, the PerfExpert output from Figure 1 results in the optimization recommendations shown in Figure 3, which include reordering

of the loops in the loop nest. The recommendation for loop blocking arises due to the high rate of L2 misses. Section 2 provides more detail and shows how the complete selection and optimization process is applied to this triply-nested loop. As we shall see later in the examination of real application codes, the appropriate optimizations are sometimes much less straightforward to identify.

Some of the suggestions include compiler flags (Figure 7). Since compiler flags depend on the compiler used, AutoSCOPE allows the user to choose among different compilers. It further allows turning on and off the inclusion of the code examples and compiler flags in the output.

We evaluated AutoSCOPE on three large-scale HPC application programs, including one that is used as a standard performance benchmark, on a supercomputing cluster at TACC. It was, almost without exception, successful in identifying and recommending the optimizations that were thought to be most appropriate by human performance experts that had tuned these codes before AutoSCOPE existed. AutoSCOPE and PerfExpert are freely available at <http://www.tacc.utexas.edu/perfexpert/>.

2. Filtering and ranking approach

This section explains how AutoSCOPE makes its recommendations. It uses the same process for each code section in PerfExpert’s output. The purpose of filtering is to eliminate inapplicable suggestions. The purpose of ranking is to order the suggestions so that the most relevant recommendations can be identified and outputted. The ultimate goal is to make the final list neither too long nor too short and, of course, to include the most appropriate optimization suggestions for each code section. Additionally, we want AutoSCOPE and its database to be easily extensible.

1. recommendation: use smaller types (e.g., float instead of double, short instead of int) categories: data-TLB, data-L2+memory-accesses attributes: -
2. recommendation: move loop invariant memory accesses out of loop categories: data-L1-accesses attributes: loop
3. recommendation: change the order of loops categories: data-TLB, data-L2+memory-accesses attributes: loop
4. recommendation: employ loop blocking and interchange categories: data-TLB, data-L2+memory-accesses attributes: loop
5. recommendation: fuse multiple loops that access the same data categories: data-L2+memory-accesses attributes: loop, multiple_loops
6. recommendation: componentize loops by factoring them into their own subroutines categories: data-L2+memory-accesses attributes: loop
7. recommendation: apply loop fission so every loop accesses just two different arrays categories: data-memory-accesses attributes: loop
8. recommendation: move loop invariant computations out of loop categories: FP-instructions attributes: loop

Figure 4: Simplified excerpt from AutoSCOPE’s recommendation database

To support ranking and filtering, each entry in the optimization database is annotated with a set of *attributes* and a non-empty set of *categories*. The attributes specify conditions that must be met for the corresponding entry to be useful. For example, an attribute might state that this is a loop

optimization and therefore only applies to loops. In contrast, the categories are used to compute weighted averages of LCPI values for the ranking. Figure 4 shows a small excerpt from the current optimization database, excluding code examples and compiler flags. We use this sample database throughout this section, along with the PerfExpert output from Figure 1, to illustrate the filtering and ranking process of AutoSCOPE, which entails the following steps.

Function versus loop filtering: AutoSCOPE uses a disjoint set of recommendations for code sections that represent loops and code sections that represent functions. This separation is a consequence of how HPCToolkit [24], upon which PerfExpert is built, treats functions and loops. Whereas functions and loops are assessed individually and do not include measurements from other functions they call, a function's assessment always includes the loops executed by this function. Hence, for any loop that is listed by PerfExpert, the enclosing function is necessarily also listed, and all suggested optimizations for the loop also apply to the function, leading to unnecessary duplication of suggestions. If, however, a function is listed but a contained loop is not, then that loop is not important and therefore loop optimizations should not be suggested for this function. Combining these two cases, we find that there is never a good reason to suggest loop optimizations for functions or function optimizations for loops. Consequently, AutoSCOPE only emits suggestions with the loop attribute for loops and suggestions without the loop attribute for functions. For example, entry 1 in the above database does not have the loop attribute and is therefore not recommended for our example loop nest.

Multiple function or loop filtering: Several of the optimizations in the database only apply if there are multiple important functions or loops. For instance, recommendations that require the reordering of functions or loops as well as recommendations to fuse multiple loops belong to this category. AutoSCOPE only makes such recommendations if multiple functions or multiple loops are included in PerfExpert's output. This is why it excludes entry 5.

Weighted LCPI ranking: Once the database entries with suitable attributes have been identified, they are ranked based on their category annotations. This is done by taking the LCPI values of the listed categories of each entry and adding them up. For instance, if a code section has a data access LCPI of 14.6 and a data TLB LCPI of 9.9, optimizations that help with data access bottlenecks will be ranked higher than optimizations that alleviate data TLB issues, but optimizations that help with both problems are ranked highest. In our example, AutoSCOPE computes the following weights: entry 3 = 22.7, entry 4 = 22.7, entry 6 = 12.8, entry 7 = 11.9, entry 8 = 3.0, and entry 2 = 1.7.

Ranking-based filtering: Because the ranking not only orders the suggestions but also assigns a metric of relevance to them, AutoSCOPE is able to filter out recommendations that are unlikely to be relevant. It currently uses a 30% threshold for this purpose, i.e., suggestions whose relevance is

less than 30% of the most relevant recommendation are discarded. 30% of 22.7 is 6.81, so entries 8 and 2 are considered not relevant enough and are filtered out.

Attribute-based tie breaking: Because there are over an order of magnitude more recommendations in our (growing) database than there are categories, the weighted LCPI ranking often yields multiple suggestions with the same rank. To break these ties, recommendations with more attributes (that must all match) are given priority. The intuition behind this approach is that an entry with more attributes is more specific and therefore more likely to be a good match for the given code section.

Order-based tie breaking: If there are still ties left, AutoSCOPE uses the order in which the suggestions are listed in the database as the final tie breaker. This allows the database writer to indicate which optimizations should be listed first in case of a tie without having to resort to additional annotations. In our example, entry 3 will be listed before entry 4 even though both of them have the same ranking (see below for why this is a good order).

Number-based filtering: This last step is optional and not enabled in the current version of AutoSCOPE. If too many suggestions are left, only the top k will be outputted, where k is a user selectable threshold. The purpose of this step is simply to curtail the list to no more than k suggestions so as not to clutter the output.

Using the above ranking and filtering approach for the loop nest assessed in Figure 1 results in the recommendations shown in Figure 3, both for the full database as well as for the database shown in Figure 4. Even without knowing what this loop nest does, it makes sense to exclude entry 1 because it is not restricted to loops and will therefore be recommended for the function containing the loop nest, entry 2 because the L1 data cache does not represent the major performance bottleneck, entry 5 because we only have a single loop nest, and entry 8 because floating-point operations are not the major performance bottleneck. It also makes sense to recommend entry 3 over entry 4 because entry 3 requires only two nested loops whereas entry 4 requires three nested loops, making entry 3 more likely to apply in general (PerfExpert does not report the nesting depth to AutoSCOPE). Entry 6 only helps with L2 data cache and data memory access problems whereas entries 3 and 4 additionally help with data TLB problems. Thus, entries 3 and 4 should be listed before entry 6. Finally, entry 7 addresses DRAM page conflicts and thus primarily helps with memory issues, which is why it is listed last.

2.1 Suggestion relevance

Applying the first transformation suggested by AutoSCOPE in Figure 3 results in the code shown in Figure 5. This optimization is simple. It improves the performance by a factor of 2.5 and eliminates all TLB problems.

```

for (i = 0; i < n; i++)
  for (k = 0; k < n; k++)
    for (j = 0; j < n; j++)
      c[i][j] += a[i][k] * b[k][j];

```

Figure 5: Matrix-matrix multiplication code after exchanging the j and k loops

Applying the second transformation from Figure 3 yields the code shown in Figure 6. As this example illustrates, applying some optimizations is quite complex and the code examples are essential to show the programmer what needs to be done. In this case, all the user has to do after modifying the code appropriately is find a good value for the parameter s , which can be done using manual trials or autotuning. Applying this optimization eliminates the remaining data-access bottlenecks and improves performance by a factor of 5.2. At this point, the performance is very good and no further optimizations are needed. Hence, AutoSCOPE not only correctly identified useful code optimizations but also eliminated a large number of irrelevant or inapplicable recommendations from consideration, thus helping the user by focusing his or her attention on just a few optimizations that are targeted for this code section.

```

for (k = 0; k < n; k += s)
  for (j = 0; j < n; j += s)
    for (i = 0; i < n; i++)
      for (kk = k; kk < k + s; kk++)
        for (jj = j; jj < j + s; jj++)
          c[i][jj] += a[i][kk] * b[kk][jj];

```

Figure 6: Matrix-matrix multiplication code after applying loop blocking

2.2 Extensibility

Adding database entries is simple. The database is stored in plain text format. New entries can be added using any text editor. Of course, the new entries need to include proper category and attribute annotations. Using plain text also makes it easy to update existing entries, such as modifying a code example, adding compiler flags, or altering attributes.

Adding extra categories is more involved. The database entries that correspond to the new category need to be annotated accordingly and the parser in AutoSCOPE has to be extended to recognize the new category. However, the rest of AutoSCOPE's functionality, such as computing the weighted average of the LCPI values and displaying the results on a web page, can be reused.

Adding new attributes requires the affected database entries' annotations to be updated and AutoSCOPE to include a new function to recognize or compute the new attribute. For example, future versions of PerfExpert might output the loop nesting depth, which is currently not available. This would allow us, for instance, to annotate entry 3 in our sample database to require at least a doubly-nested loop. Such an annotation would prevent AutoSCOPE from recommending entry 3 for any non-nested loops.

3. Related work

There are many performance evaluation tools with a wide range of approaches and functionalities. Performance tools may implement four functions: measurement, analysis based on the measurements, recommendation of source-code optimizations, and automation of source-code optimizations. The tools can be further classified by the basis for measurement and analysis: performance-counter-based versus event-trace-based and by whether the tool requires the source code to be annotated to generate measurements. Since the subject of this paper is derivation of recommendations for source-code optimizations for bottlenecks that are identified and characterized through performance-counter measurements and analyses, we only briefly mention papers that do not, in our best judgment, consider source-code optimizations or where the analyses are based on event traces.

Tau [20], [25], PerfSuite [16], [22], HPCToolkit [12], [24], IPM [14], and OpenSpeedShop [19] are among the most powerful and widely used tools that provide performance-counter-based measurement and analysis. Each of these tools provides flexible and in-depth measurement and association of performance bottlenecks with source-code segments. Each tool provides the measurements and at least some of the analyses upon which optimization can be based but do not extend to recommendation of or automation of source-code optimizations.

PerfExplorer [13] extends Tau with additional analysis and diagnostic capabilities. However, PerfExplorer/Tau uses code instrumentation and event tracing, which can perturb the execution behavior, and does not recommend source-code optimizations. In contrast, PerfExpert bases its analysis and optimization recommendations upon data provided by HPCToolkit, which requires no code instrumentation and uses CPU performance counters to minimize perturbation.

There are several tools that provide source-code optimizations for some types of bottlenecks. ThreadSpotter [26] captures information about data access patterns from a cache simulator and offers advice on related losses, specifically latency exposed due to poor locality, competition for bandwidth, and false sharing. It recommends possible optimizations for bottlenecks resulting from data accesses. PerfExpert attempts a more comprehensive diagnosis of bottlenecks, targeting not only data locality but also instruction locality, floating-point performance, etc. and recommends optimizations across this spectrum. ThreadSpotter does not attempt automated optimizations.

SLO [2] uses cache profiling to measure data reuse distances and other locality metrics. It associates these measures with code segments, particularly loops, and suggests optimizations such as loop tiling or loop interchange.

Paradyn [17], based on Dyninst [3], is a performance measurement tool for parallel and distributed programs. Performance instrumentation is inserted into the application and modified during execution. It associates bottlenecks

with specific causes and program parts but does not extend to recommending application optimizations.

Periscope [11] collects and aggregates performance information through an agent-based approach. It provides identification of the source-code locations of performance bottlenecks and analyses of causes for the bottlenecks.

MAQAO [9] is a performance analysis tool that combines performance counter measurements with static information to generate diagnoses. It derives the static information from the assembly code. It contains a knowledge base of important assembly patterns, which can be associated with hints for possible code optimizations.

The IBM Productive, Easy-to-use, Reliable Computing System (PERCS) project [5], [6], [27] is building an automated system that detects and analyzes performance bottlenecks in application codes, identifies potential source-code optimizations, and includes automated optimization capabilities. The Bottleneck Detection Engine (BDE), which is the core of the framework, utilizes a database of rules to detect bottlenecks in the given application. The BDE feeds the information on bottleneck locations, including metrics associated with the bottlenecks, to the user. It may also suggest how much improvement could be obtained by the optimization of a given bottleneck. In addition to suggestions to the user, IBM's tool also supports directly modifying the source code and applying standard transformations through the compiler. The limitation of PERCS is that it requires the use of IBM's proprietary software stack including its compilers.

Systems that base their analyses on event traces include KOJAK [18], KappaPI [10] and the Parallel Performance Wizard [23]. KOJAK aims at the development of a generic automatic performance analysis environment for parallel programs. The Parallel Performance Wizard attempts automatic diagnosis as well as automated optimization. It is, however, based on event trace analysis and requires program instrumentation. Its primary applications have been problems associated with the partitioned global address space (PGAS) programming model, although it applies to other performance bottleneck issues as well.

Cray's ATEExpert [15] graphically displays the performance of parallel programs. It points the user to specific problem areas in the source code, tries to explain why the problems are occurring, and suggests steps to resolve them. It does not provide code templates or rank suggestions.

The Performance Engineering Research Institute (PERI) Autotuning project [1] combines measurement and search-directed auto-tuning in a multistep process to obtain automated optimization. It can be viewed as a special case of an expert system where one flexible solution method is applied to all types of bottlenecks. It is unclear whether autotuning by itself can effectively optimize the wide spectrum of bottlenecks that arise when executing complex codes on multi-core chips and multi-socket nodes. We hope to be able to incorporate methods from this project in the optimization capabilities of a future version of PerfExpert.

4. Evaluation methodology

4.1 System

We used the PerfExpert installation on Ranger, the supercomputing Sun Constellation Linux Cluster at the Texas Advanced Computing Center. Ranger contains 3,936 16-way SMP compute nodes made of 15,744 quad-core 2.3 GHz AMD Opteron (Barcelona) processors, i.e., 62,976 compute cores. It has 123 TB of main memory, 1.7 PB of global disk space, and a theoretical peak performance of 579 TFLOPS. All compute nodes are interconnected using InfiniBand in a seven-stage full-CLOS fat-tree topology providing 1 GB/s point-to-point bandwidth.

4.2 Applications

We have tested AutoSCOPE on MANGLL/DGADVEC, HOMME, and LIBMESH/EX18. These large-scale HPC programs represent various application domains. They were all compiled with the Intel compiler v10.1.

MANGLL is a scalable adaptive high-order discretization library. It supports dynamic parallel adaptive mesh refinement and coarsening, which is essential for numerical solution of the partial differential equations (PDEs) arising in many multiscale physical problems. DGADVEC is an application built on top of MANGLL for the numerical solution of the energy equation that is part of the coupled system of PDEs arising in convection simulations, describing the viscous flow and temperature distribution in Earth's mantle. MANGLL and DGADVEC are written in C.

HOMME is an atmospheric general circulation model consisting of a dynamical core based on the hydrostatic equations, coupled to sub-grid scale models of physical processes. The HOMME code is designed to provide 3D global atmospheric simulation similar to the Community Atmospheric Model. The benchmark version of HOMME we are using was one of NSF's acceptance benchmark programs for Ranger. It is written in Fortran 95.

The LIBMESH library provides a framework for the numerical approximation of partial differential equations using continuous and discontinuous Galerkin methods on unstructured hybrid meshes. It supports parallel adaptive mesh refinement computations as well as 1D, 2D, and 3D steady and transient simulations on a variety of popular geometric and finite element types. EX18 uses LIBMESH to solve the transient nonlinear problem using the object-oriented FEMSystem class framework. LIBMESH and EX18 are written in C++.

5. Results

This section compares the suggestions produced by AutoSCOPE with actual optimizations that performance experts implemented to accelerate large-scale HPC codes. Due

to space reasons, we only show one key loop per program.

Figure 7 shows the recommendations AutoSCOPE makes for the most important loop in LIBMESH/EX18. This loop performs a large number of memory accesses, most of which hit in the L1 data cache, and quite a few floating-point operations. Hence, AutoSCOPE focuses on suggestions that reduce the number of load instructions, boost the bandwidth to the L1 data cache, and reduce the number of floating-point instructions.

Loop in function <code>NavierSystem::element_time_derivative(...)</code> (23.3% of runtime)
move loop invariant memory accesses out of loop <code>loop i {a[i] = b[i] * c[j];} → temp = c[j]; loop i {a[i] = b[i] * temp;}</code>
enable the use of vector instructions to transfer more data per access align arrays, use only stride-one accesses, make loop count even (pad arrays) <code>struct {double a, b;} s[63]; for (i = 0; i < 63; i++) {s[i].a = 0; s[i].b = 0;} → __declspec(align(16)) double a[64], b[64]; for (i = 0; i < 64; i++) {a[i] = 0; b[i] = 0;} use the “-opt-streaming-stores always” compiler flag</code>
move loop invariant computations out of loop <code>loop i {x = x + a * b * c[i];} → temp = a * b; loop i {x = x + temp * c[i];}</code>

Figure 7: Condensed AutoSCOPE recommendations for the most important loop in EX18

When we manually optimized this code (before AutoSCOPE existed), we obtained a substantial speedup by applying the first and third recommendation, i.e., by factoring out common subexpressions involving memory accesses and by moving loop invariant code [4]. Based on simple tests, the author of EX18 had assumed that the compiler would do this automatically. However, several of the common subexpressions we found involve C++ templates and most of them use pointer indirections, which seem to make the code too complex for the compiler to optimize. These simple optimizations (for a human) made the loop 32% faster, yielding an application-wide speedup of 5%.

Figure 8 shows AutoSCOPE’s recommendations for one of the two key loops in DGADVEC. This loop’s performance profile is quite similar to that of the EX18 loop discussed above except it performs significantly more floating-point operations. Hence, the recommendations are similar but the order in which they are listed is different.

Loop in function <code>dgadvecRHS()</code> at <code>dgadvec.c:993</code> (19.4% of total runtime)
move loop invariant computations out of loop <code>loop i {x = x + a * b * c[i];} → temp = a * b; loop i {x = x + temp * c[i];}</code>
componentize loops by factoring them into their own subroutines <code>... loop i {...} ... loop j {...} ... → void li() {...}; void lj() {...}; ... li(); ... lj(); ...</code>
move loop invariant memory accesses out of loop <code>loop i {a[i] = b[i] * c[j];} → temp = c[j]; loop i {a[i] = b[i] * temp;}</code>
enable the use of vector instructions to transfer more data per access align arrays, use only stride-one accesses, make loop count even (pad arrays) <code>struct {double a, b;} s[63]; for (i = 0; i < 63; i++) {s[i].a = 0; s[i].b = 0;} → __declspec(align(16)) double a[64], b[64]; for (i = 0; i < 64; i++) {a[i] = 0; b[i] = 0;} use the “-opt-streaming-stores always” compiler flag</code>

Figure 8: Condensed AutoSCOPE recommendations for the most important loop in DGADVEC

Together with the developers of DGADVEC, we have been able to accelerate this loop through vectorization. The

primary performance problem is the L1 load-to-use hit latency of three cycles, which cannot be hidden as there are not enough independent instructions available to execute. Since this latency is fixed in hardware, we can only reduce the average latency by increasing the bandwidth, i.e., accessing multiple data items per memory transaction through the use of SSE instructions. Hence, we rewrote the loop so that the compiler can vectorize it [8] (i.e., we applied the fourth recommendation). Comparing the old and new loop implementations, we found that the number of executed instructions is 44% lower and the number of L1 data-cache accesses is 33% lower due to the vectorization [4].

Figure 9 shows the optimization suggestions for an important loop in HOMME, which has also been manually tuned. This loop suffers primarily from bad memory access performance. Many of the accesses miss in all cache levels and go to main memory. As a consequence, AutoSCOPE recommends optimizations that aim at helping the compiler optimize the code better, enhance the memory access patterns, and improve the main memory latency.

Loop in function <code>preq_robert()</code> at <code>prim_si_mod.F90:846</code> (8.9% of total runtime)
componentize loops by factoring them into their own subroutines <code>... loop i {...} ... loop j {...} ... → void li() {...}; void lj() {...}; ... li(); ... lj(); ...</code>
change the order of loops <code>loop i { loop j {...} } → loop j { loop i {...} }</code>
employ loop blocking and interchange <code>loop i { loop k { loop j { c[i][j] = c[i][j] + a[i][k] * b[k][j];}} } → loop k step s { loop j step s { loop i { for (kk = k; kk < k + s; kk++) { for (jj = j; jj < j + s; jj++) {c[i][jj] = c[i][jj] + a[i][kk] * b[kk][jj];}}}} }</code>
apply loop fission so every loop accesses just two different arrays <code>loop i {a[i] = a[i] * b[i] - c[i];} → loop i {a[i] = a[i] * b[i];} loop i {a[i] = a[i] - c[i];}</code>
move loop invariant computations out of loop <code>loop i {x = x + a * b * c[i];} → temp = a * b; loop i {x = x + temp * c[i];}</code>

Figure 9: Condensed AutoSCOPE recommendations for an important loop in HOMME

A performance expert has successfully sped up this loop by applying microfission (the fourth recommendation) to reduce DRAM page conflicts, which makes the main memory accesses substantially faster [4], [7]. On a Ranger node, only 32 DRAM pages can be open at once. With 16 threads running on the 16 cores of a node, each thread can access at most two different memory areas simultaneously without losing performance. Thus, applying microfission so that each loop only processes two arrays eliminates DRAM page conflicts. However, because the compiler automatically fuses the loops, it was necessary to also break out each loop into a separate procedure (the first recommendation), which results in a 62% speedup and much better core utilization.

6. Conclusion and future work

AutoSCOPE helps programmers by automatically recommending source-code optimizations and compiler flags for alleviating node-level performance bottlenecks that have been identified by the PerfExpert measurement and analysis

tool. AutoSCOPE processes PerfExpert's output using a set of rules to identify matching recommendations in its annotated database. It then ranks these recommendations to select the most appropriate ones. AutoSCOPE is constructed as an extensible framework to which we can add annotations and rules to extend its capabilities or to adapt it to different execution environments. Our evaluation on real HPC applications has demonstrated almost 100% conformance to human expert optimization selections. While the output sometimes still includes inapplicable suggestions, AutoSCOPE correctly eliminates over 95% of the suggestions from the database that do not apply, thus helping the user a great deal by focusing his or her attention on just a few targeted optimizations. In future work, we plan to add more rules and annotations to further improve the selection quality and to apply selected source-code optimizations automatically for straightforward cases.

7. References

- [1] D. Bailey, J. Chame, C. Chen, J. Dongarra, M. Hall, J. Hollingsworth, P. Hovland, S. Moore, K. Seymour, J. Shin, A. Tiwari, S. Williams, and H. You. "PERI Auto-Tuning." *Journal of Physics: Conference Series*, 125(1):012089, 2008.
- [2] K. Beyls and E. D'Hollander. "Refactoring for Data Locality." *IEEE Computer*, Vol. 42, no. 2, pp. 62-71. 2009.
- [3] B. R. Buck and J. K. Hollingsworth. "An API for Runtime Code Patching." *Journal of High Performance Computing Applications*, 14:317-329. 2000.
- [4] M. Burtscher, B.D. Kim, J. Diamond, J. McCalpin, L. Koesterke, and J. Browne. "PerfExpert: An Easy-to-Use Performance Diagnosis Tool for HPC Applications." *SC 2010 Int. Conference for High-Performance Computing, Networking, Storage and Analysis*. November 2010.
- [5] Chung, G. Cong, D. Klepacki, S. Sbaraglia, S. Seelam, and H-F. Wen. "A Framework for Automated Performance Bottleneck Detection." *13th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments*. 2008.
- [6] G. Cong, I-H. Chung, H. Wen, D. Klepacki, H. Murata, Y. Negishi, and T. Moriyama. "A Holistic Approach towards Automated Performance Analysis and Tuning." *Euro-Par 2009*. 2009.
- [7] J. Diamond, M. Burtscher, J. McCalpin, B.D. Kim, S. Kecker, and J. Browne. "Making Sense of Performance Counter Measurements on Supercomputing Applications." *2011 IEEE International Symposium on Performance Analysis of Systems and Software*. April 2011.
- [8] J. Diamond, B.D. Kim, M. Burtscher, S. Keckler, K. Pingali, and J. Browne. "Multicore Optimization for Ranger." *2009 TeraGrid Conference*. June 2009.
- [9] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby. "Exploring Application Performance: a New Tool for a Static/Dynamic Approach." *The Sixth Los Alamos Computer Science Institute Symp*. 2005.
- [10] Antonio Espinosa, Tomas Margalef, and Emilio Luque. "Automatic detection of parallel program performance problems." *SIGMETRICS Symposium on Parallel and Distributed Tools*, p. 149. 1998.
- [11] M. Gerndt and M. Ott. "Automatic performance analysis with Periscope." *Concurrency Computation: Practice and Experience*. 2009.
- [12] HPCToolkit: <http://www.hpctoolkit.org/>. Last accessed April 1, 2011.
- [13] K. A. Huck, A. D. Malony, S. Shende, and A. Morris. "Knowledge Support and Automation for Performance Analysis with PerfExplorer 2.0." *Large-Scale Programming Tools and Environments, Special Issue of Scientific Programming*, vol. 16, no. 2-3, pp. 123-134. 2008.
- [14] IPM: <http://ipm-hpc.sourceforge.net/>. Last accessed April 1, 2011.
- [15] J. Kohn and W. Williams. "ATExpert." *Journal of Parallel and Distributed Computing*, 18:2, pp. 205-222. 1993.
- [16] Rick Kufrin. "PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux." *6th Int. Conference on Linux Clusters: The HPC Revolution*. 2005.
- [17] 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*, 28:37-46. 1995.
- [18] B. Mohr and F. Wolf. "KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications." *Int. Conference on Parallel and Distributed Computing*. 2003.
- [19] OpenSpeedShop: <http://www.openspeedshop.org/wp/>. Last accessed April 1, 2011.
- [20] S. Shende and A. Malony. "The Tau Parallel Performance System." *International Journal of High Performance Computing Applications*, 20(2): 287-311.
- [21] PerfExpert: <http://www.tacc.utexas.edu/perfexpert/>. Last accessed April 1, 2011.
- [22] PerfSuite: <http://perfsuite.ncsa.uiuc.edu/>. Last accessed April 1, 2011.
- [23] H-H. Su, M. Billingsley, and A. D. George. "Parallel Performance Wizard: A Performance Analysis Tool for Partitioned Global-Address-Space Programming." *9th Int. Workshop on Parallel & Distr. Scientific and Engineering Computing*. 2008.
- [24] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M.W. Fagan, and M. Krentel. "HPCToolkit: performance tools for scientific computing." *Journal of Physics: Conference Series*, 125. 2008.
- [25] Tau: <http://www.cs.uoregon.edu/research/tau/home.php>. Last accessed April 1, 2011.
- [26] ThreadSpotter: <http://www.roguewave.com/products/threadspotter.aspx>. Last accessed April 1, 2011.
- [27] H. Wen, S. Sbaraglia, S. Seelam, I. Chung, G. Cong, and D. Klepacki. "A productivity centered tools framework for application performance tuning." *Fourth International Conference on the Quantitative Evaluation of Systems*, pp. 273-274. 2007.

MapReduce with Deltas

R. Lämmel and D. Saile

Software Languages Team, University of Koblenz-Landau, Germany

Abstract—The MapReduce programming model is extended conservatively to deal with deltas for input data such that recurrent MapReduce computations can be more efficient for the case of input data that changes only slightly over time. That is, the extended model enables more frequent re-execution of MapReduce computations and thereby more up-to-date results in practical applications. Deltas can also be pushed through pipelines of MapReduce computations. The achievable speedup is analyzed and found to be highly predictable. The approach has been implemented in Hadoop, and a code distribution is available online. The correctness of the extended programming model relies on a simple algebraic argument.

Keywords: MapReduce; Delta; Distributed, Incremental Algorithms

1. Introduction

We are concerned with the MapReduce programming model [1], which is widely used for large-scale data processing problems that can benefit from massive data parallelism. MapReduce is inspired by functional programming idioms, and it incorporates specific ideas about indexing and sorting; see [2] for a discussion of the programming model. There exist several proprietary and open-source implementations that make MapReduce available on different architectures.

Research question

The problem of crawling the WWW may count as the archetypal application of MapReduce. A particular crawler may operate as follows: web sites are fetched; outlinks are extracted; accordingly, more web sites are fetched in cycles; a database of inverse links (“inlinks”) is built to feed into page ranking; eventually, an index for use in web search is built; see Fig. 1 for the corresponding workflow.

In many MapReduce scenarios (including the one of crawling and indexing), the question arises whether it is possible to achieve a speedup for recurrent executions of a MapReduce computation by making them incremental.

A crawler is likely to find about the same pages each time it crawls the web. Hence, the complete re-computation of the index is unnecessarily expensive, thereby limiting the frequency of re-executing the crawler as needed for an up-to-date index. A more up-to-date index is feasible if the index is incrementally (say efficiently) updated on the grounds of the limited changes to the crawl results.

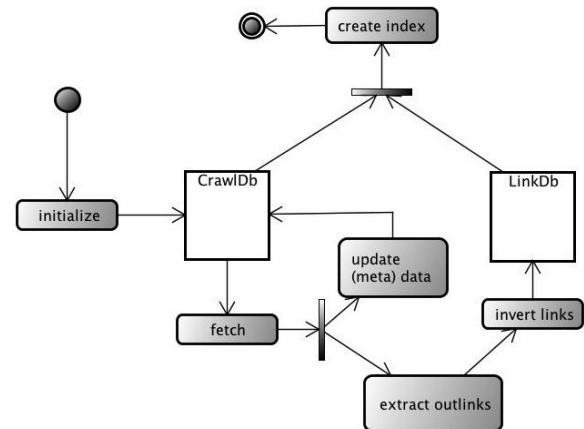


Fig. 1: Workflow of a simple crawler with indexing

Contributions

- The MapReduce programming model is enhanced to explicitly incorporate deltas of inputs of recurrent MapReduce computations. This enhancement is based on a simple algebraic insight that has not been exploited elsewhere.
- Based on benchmarks for delta-aware MapReduce computations, it is found that deltas are of limited use when used naively, but they provide substantial, predictable speedups—when applying specific techniques for computing deltas and merging them with previous results.

Our implementation and corresponding measurements are based on Apache’s Hadoop [3]—an open-source implementation of MapReduce which targets clusters of networked computers with a distributed file system. A code distribution is available online through the paper’s website.¹

Road-map

Sec. 2 expands on the introductory problem of crawling and indexing, thereby clarifying the motivation of our work. Sec. 3 rehashes MapReduce in a way that is specifically suitable for initiating a discussion of deltas. Sec. 4 extends the MapReduce programming model to incorporate deltas. Sec. 5 discusses different options for computing deltas. Sec. 6 defines and executes benchmarks for delta-aware MapReduce computations. Sec. 7 discusses related work. Sec. 8 concludes the paper.

2. Motivation

Crawling without deltas Any search engine relies on one or more indexes that are computed from information that is

¹<http://softlang.uni-koblenz.de/deltamr>

obtained by web crawls. A typical crawler, such as *Nutch* [4], performs several tasks that can be implemented as a *pipeline* of MapReduce jobs; we refer again to Fig. 1 for a simple workflow for crawling and indexing. The crawler maintains a database, *CrawlDb*, with (meta) data of discovered websites. Before crawling the web for the first time, *CrawlDb* is initialized with seed URLs. The crawler performs several cycles of fetching. In each cycle, a *fetch list* (of URLs) is obtained from *CrawlDb*. The corresponding web sites are downloaded and *CrawlDb* is updated with a time stamp and other data. Further, the crawler extracts outlinks and aggregates them in *LinkDb* so that each URL is associated with its inlinks. The resulting reverse web-link graph is useful, for example, for ranking sites such as with *PageRank* [5]. Eventually, *CrawlDb* and *LinkDb* are used to create an index, which can be queried by a search engine.

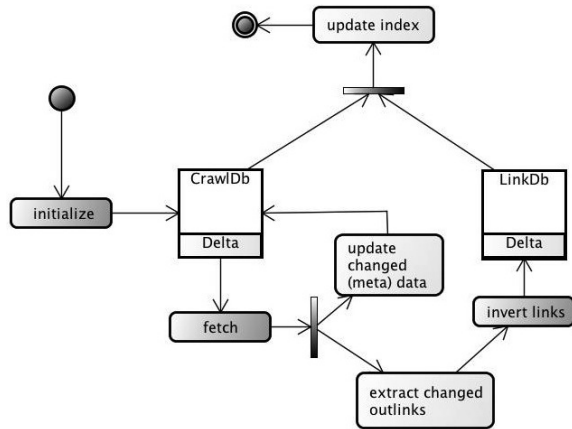


Fig. 2: Crawler using deltas

Crawling with deltas Suppose only a small fraction of all web sites changes. Then it can be more efficient to determine those changes (say, “deltas”) and to update the index accordingly. Fig. 2 revises the simple crawler from Fig. 1 so that deltas are used in several positions. That is, in each crawl cycle, a delta of changed sites is determined and corresponding deltas of outlinks, *CrawlDb*, and *LinkDb* are derived so that the index can be updated incrementally.

3. A simple view on MapReduce

For the rest of the paper, we will not deal with the complex scenario of crawling and indexing. We resort to “the problem of counting the number of occurrences of each word in a large collection of documents” [1]. In sequential, imperative (pseudo) code, the problem is solved as follows:

<pre> Input: a collection of uri-document pairs <i>c</i> Output: a map <i>m</i> from words to counts Algorithm: for each $\langle u, d \rangle$ in <i>c</i> do for each <i>w</i> in <i>words</i>(<i>d</i>) do $m[w] = m[w] + 1$; // <i>m</i>[<i>w</i>] is initially 0. </pre>

Fig. 3: Sequential, imperative word-occurrence count

This direct approach does not stipulate massive parallelism for iterating over *c* because of the use of a global data structure for the map (say, dictionary) *m*. The aspects of data extraction and reduction are to be separated. Extraction is supposed to produce a stream of word-occurrence counts as follows:

<pre> Input: a collection of uri-document pairs <i>c</i> Output: a stream <i>s</i> of words-occurrence counts Algorithm: for each $\langle u, d \rangle$ in <i>c</i> do for each <i>w</i> in <i>words</i>(<i>d</i>) do yield $\langle w, 1 \rangle$; // per-document extraction </pre>
--

Fig. 4: Extraction amenable to parallelism and distribution

(The role of the boxed code is explained in a second.) In general, extraction returns a stream of key-value pairs to be reduced eventually (see below). In the example, words are keys and counts are values. The intermediate stream can be produced in a massively parallel manner such that input partitions are assigned to nodes in a cluster of machines. Subject to a distributed file system, the partitions may be readily stored with the nodes that process them.

Reduction requires grouping of values by key:

<pre> Input: a stream <i>s</i> of key-value pairs Output: a map (say, a dictionary) <i>m'</i> of key-list pairs Algorithm: for each $\langle k, v \rangle$ in <i>s</i> do $m'[k] = \text{append}(m'[k], v)$; // <i>m'</i>[<i>k</i>] is initially the empty list. </pre>

Fig. 5: Group key-value pairs

Reduction commences as follows:

<pre> Input: a map <i>m'</i> of key-list pairs Output: a map <i>m</i> from words to counts Algorithm: for each $\langle k, g \rangle$ in <i>m'</i> do { $r = 0$; for each <i>v</i> in <i>g</i> do // per-key reduction $r = r + v$; $m[k] = r$; } </pre>
--

Fig. 6: Reduction amenable to parallelism and distribution

(The role of the boxed code is explained in a second.) Grouping and reduction can be distributed (parallelized) by leveraging the fact that the key domain may be partitioned.

The original sequential description of Fig. 3 is much more concise than the sliced, parallelism-enabling development of Fig. 4–6. However, it is easy to realize that most of the code is problem-independent. In fact, the only problem-specific code is the one that is boxed in Fig. 4 and Fig. 6. That is, the first box covers data extraction at a fine level of granularity; the second box covers data reduction per intermediate key. In practice, MapReduce computations are essentially specified in terms of two functions *mapper* and *reducer*:

```

function mapper(u, d) {
  for each w in words(d) do
    yield (w, 1);
}
function reducer(k, g) {
  r = 0;
  for each v in g do r = r + v;
  return r;
}

```

Fig. 7: The functionality for word-occurrence counting

Summary MapReduce computations extract intermediate key-value pairs from collections of input documents or records. Such extraction can be easily parallelized if input data is readily partitioned to reside on machines in a cluster. The resulting intermediate key-value pairs are to be grouped by key. The key domain is partitioned so that parallelism can be applied for the reduction of values per key. MapReduce implementations allow the specification of the number of mapper and reducer nodes as well the specification of a *partitioner* that associates partitions of the intermediate key domain with reducers.

4. MapReduce with deltas

Deltas The input for MapReduce computations is generally a keyed collection, in fact, an ordered list [1]. Given two generations of input data i and i' , a delta $\Delta_{i,i'}$ can be defined as a quadruplet of the following sub-collections:

- $\Delta_{i'_+}$ Part of i' with keys not present in i .
- Δ_{i_-} Part of i with keys not present in i' .
- $\Delta_{i_{\neq}}$ Part of i whose keys map to different values in i' .
- $\Delta_{i'_{\neq}}$ Part of i' whose keys map to different values in i .

The first part corresponds to added key-value pairs; the second part corresponds to removed key-value pairs; the third and fourth parts correspond to modified key-value pairs (“before” and “after”). Modification can be modeled by deletion followed by addition. Hence, we simplify $\Delta_{i,i'}$ to consist only of two collections:

$$\begin{aligned}\Delta_+ &= \Delta_{i'_+} + \Delta_{i'_{\neq}} \\ \Delta_- &= \Delta_{i_-} + \Delta_{i_{\neq}}\end{aligned}$$

The simple but important insight is that MapReduce computations can be applied to the parts of the delta and combined later with the result for i so that the result for i' is obtained more efficiently than by computing i' naively.

Algebraic requirements Correctness conditions are needed for the non-incremental and incremental execution to agree on the result. This is similar to the correctness conditions for classic MapReduce that guarantee that different distribution schedules all lead to the same result.

In the case of classic MapReduce, the mapper is not constrained, but the reducer is required to be (the iterated application of) an associative operation [1]. More profoundly, reduction is *monoidal* in known applications of

MapReduce [2], [6]. That is, reduction is indeed the iterated application of an associative operation “ \bullet ” with a unit u . In the case of the word-occurrence count example, reduction iterates *addition* “ $+$ ” with “0” as unit. The parallel execution schedule may be more flexible if commutativity is required in addition to associativity [2].

Additional algebraic constraints are needed for MapReduce computations with deltas. That is, we require an *Abelian group*, i.e., a monoid with commutativity for “ \bullet ” and an operation “ $\bar{\cdot}$ ” for an inverse element such that $x \bullet \bar{x} = u$ for all x . In the case of the word-occurrence count example, addition is indeed commutative, and the inverse element is to be determined by negation. Hence, we assume that MapReduce computations are described by two ingredients:

- A mapper function—as illustrated in Fig. 7.
- An Abelian group—as a proxy for the reducer function.

MapReduce computations with deltas We are ready to state a law (without proof) for the correctness of MapReduce computations with deltas. Operationally, the law immediately describes how the MapReduce result for i needs to be updated by certain MapReduce results for the components of the delta so that the MapReduce result for i' is obtained; the law refers to “ \bullet ”—the commutative operation of the reducer:

$$\begin{aligned}MapReduce(f, g, i') &= MapReduce(f, g, i) \\ &\bullet MapReduce(f, g, \Delta_+) \\ &\bullet MapReduce(\bar{f}, g, \Delta_-)\end{aligned}$$

Here, f is the mapper function, g is an Abelian group, and \bar{f} denotes lifted inversion. That is, if f returns a stream of key-value pairs, then \bar{f} returns the corresponding stream with inverted values. In imperative style, we describe the inversion of extraction as follows:

<i>Input</i> : a stream s of key-value pairs
<i>Output</i> : a stream s' of key-value pairs
<i>Parameter</i> : an inversion operation $\bar{\cdot}$ on values
<i>Algorithm</i> :
for each $\langle k, v \rangle$ in s do
yield $\langle k, \bar{v} \rangle$; // value-by-value inversion

Fig. 8: Lifted inversion

Fig. 9 summarizes the workflow of MapReduce computations with deltas. Clearly, we assume that we can compute deltas; see the node “Compute delta”. Such deltas are then processed with the MapReduce computation such that deleted pairs are inverted; see the node “MapReduce”. One can either merge original result with the result for the delta, or one can propagate the latter to further MapReduce computations in a pipeline.

5. Computation of deltas

Deltas can be computed in a number of ways.

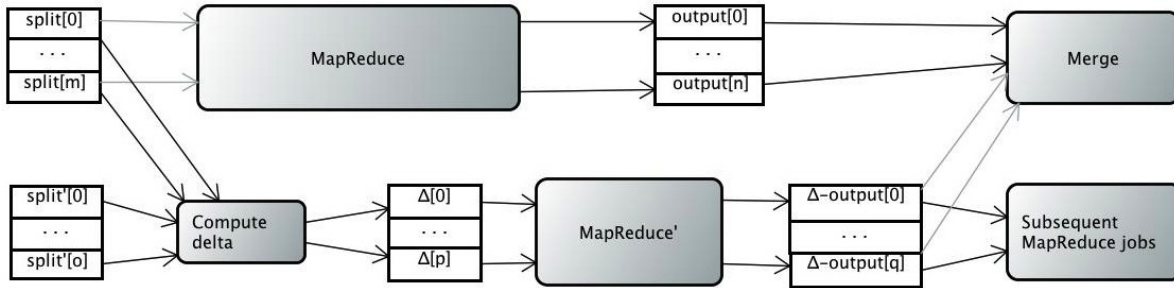


Fig. 9: MapReduce with deltas

MapReduce-based delta If we assume that both generations of input, i and i' , have been regularly stored in the distributed file system, then the delta can be computed with classic MapReduce as follows:

```

Input: the concatenated input  $append(i, i')$ 
Output: the (encoded) delta  $\Delta_{i, i'}$ 
Algorithm (MapReduce):
function mapper( $k, v$ ) {
    if  $k$  in  $i$  then  $sign := "-"$ ; else  $sign := "+"$ ;
    return  $\langle k, \langle sign, v \rangle \rangle$ ; // attach sign
}
function reducer( $k, g$ ) {
     $\langle s_1, v_1 \rangle := g.next()$ ;
    if  $\neg g.hasNext()$ 
    then return  $[\langle s_1, v_1 \rangle]$ ; // "added" or "deleted"
    else {
         $\langle s_2, v_2 \rangle := g.next()$ ;
        if  $v_1 == v_2$ 
        then return  $[\ ]$ ; // "preserved"
        else return  $[\langle s_1, v_1 \rangle, \langle s_2, v_2 \rangle]$ ; // "modified"
    }
}
    
```

Fig. 10: Computing a delta with MapReduce

The mapper qualifies the values of key-value pairs from i and i' with “-” and “+” respectively—for *potential* deletion or addition; see the condition “ k in i ” in the figure. Hadoop [3] and other implementations of MapReduce can discriminate between different input files in the map phase.

Reduction receives 1-2 values per original key depending on whether a key occurs in either i or i' or both. For simplicity, keys are assumed to be unique in each of i and i' . (Irregular cases require a slightly more advanced reduction.) In the case of a single value, a potential deletion or addition becomes definite. In the case of two values, two equal values cancel out each other, whereas two unequal values contribute to both deletion and addition.

Delta after iteration It is possible to aggressively reduce the volume of delta by exploiting a common idiom for MapReduce computations. That is, extraction is typically based on uniform, structural decomposition, say iteration. Consider the for-loop for extracting word-occurrence counts from documents—as of Fig. 7:

```

for each  $w$  in  $words(d)$  do
    yield  $\langle w, 1 \rangle$ ;
    
```

That is, the document is essentially decomposed into words from which key-value pairs are produced. Instead, the document may also be first decomposed into lines, and then, in turn, into words:

```

for each  $l$  in  $lines(d)$  do
    for each  $w$  in  $words(l)$  do
        yield  $\langle w, 1 \rangle$ ;
    
```

In general, deltas could be determined at all accessible levels of decomposition. In the example, deltas could be determined at the levels of documents (i.e., the values of the actual input), lines, and words. For the problem at hand, line-level delta appears to be useful according to established means for delta creation such as “text diff” [7]. MapReduce computations with deltas are easily configured to exploit different levels. When computing the delta, as defined in Fig. 10, the case “ $\neg(v_1 == v_2)$ ” must be refined to decompose v_1 and v_2 and to compute the delta at the more detailed level. In implementations of MapReduce, one can indeed exercise different levels. For instance, Hadoop [3] assumes that MapReduce jobs are configured with “input formatters” which essentially decompose the input files.

Delta based on map-side join Overall, the costs of MapReduce-based computation of the delta are substantial. Essentially, both generations of input have to be pumped through the network so that a reducer can cancel out matching key-value pairs. These costs would need to be matched by the savings achievable through deltas in a MapReduce computation or a pipeline.

There is a relevant MapReduce-like abstraction, which can be used to drastically reduce network communication during delta computation. That is, *map-side join* [8], [9] can be used to map over multiple inputs simultaneously such that values with the same key but from different inputs are mapped together. To this end, the inputs must be equally sorted and partitioned so that matching partitions can be dealt with simultaneously. Network communication is reduced since no reduction is involved. Network communication is completely eliminated if matching partitions are available on the same machine. (Map-side join is available, for example, in Hadoop [3].) It is often possible to meet the requirements of map-side join. For instance, a crawler may be set up to write crawling results accordingly.

Streaming delta An even more aggressive optimization is to produce and consume the second generation of input data in streaming mode. Just as before, it is necessary to assume that both generations are sorted in the same manner. Such streaming is feasible for tasks that essentially generate “sorted” data. Streaming can be also used to *fuse* two MapReduce computations—as known from functional programming [10]. Compared to all other forms of computing deltas, streaming delta does not write (and hence not read) the second generation.

6. Benchmarking

We present simple benchmarks to compare non-incremental (say, classic) and incremental (say, delta-aware) MapReduce computations. We ran the benchmarks on a university lab.² The discussion shows that speedups are clearly predictable when using our method.

TeraByte Sort TeraByte Sort (or the variation—MinuteSort) is an established benchmark to test the throughput on a MapReduce implementation when using it for *sorting* with (in one typical configuration) 100-byte records out of which 10 bytes constitute the key [11], [12], [13]. The mapper and reducer functions for this benchmark simply *copy* all data. The built-in sorting functionality of MapReduce implies that intermediate key-value pairs are sorted per reducer. The partitioner is defined to imply *total ordering* over the reducers. Hadoop—the MapReduce implementation that we use—has been a winner of this benchmark in the past.

The established implementation of TeraByte Sort (see, e.g., [12], [13]) samples keys in the input from which it builds a trie so that partitioning is fast. Instead, our partitioner does not leverage any sampling-based trie because we would otherwise experience uneven reducer utilization for MapReduce jobs on sorted data. Here we note that we must process sorted data in compound MapReduce computations; see the discussion of pipelines below. We use datatype `long` (8 bytes) for keys instead of byte sequences of length 10, thereby simplifying partitioning.

Fig. 11 shows the benchmark results for TeraByte Sort. The “incremental” version computes the delta by a variation of Fig. 10. There are also optimized, incremental versions: (map-side) “join” and “streaming”—as discussed in Sec. 5. The shown costs for the incremental versions include *all* costs that are incurred by recomputing the same result as in the non-incremental version: this includes costs of computing the delta and performing the merge. It is important to note that we implement merge by map-side join.

It is not surprising that the non-incremental version is faster than all incremental versions except for streaming.

²Cluster characteristics: we used Hadoop version 0.21.0 on a cluster of 40 nodes with an *Intel(R) Pentium(R) 4 CPU 3.00GHz* and 2 x 512MB SDRAM and 6GB available disk space. All machines are running *openSUSE 11.2* with Java version *1.6.0_24* and are connected via a *100Mbit Full-Duplex-Ethernet* network.

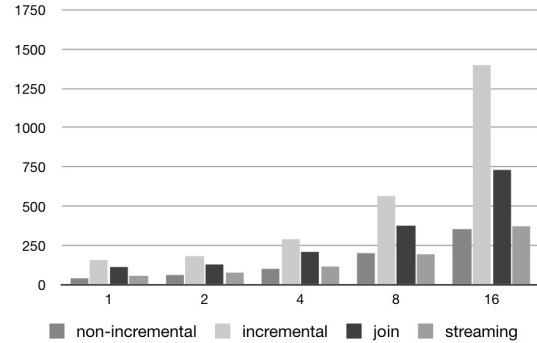


Fig. 11: Runtimes in seconds (y-axis) for non-incremental and incremental TeraByte Sort for different input sizes in GB (x-axis) where the size of the deltas for the incremental versions is assumed to be 10 % of the input size.

That is, computing a delta for data on files means that both generations are processed whereas non-incremental sorting processes only the new generation. Also, the merge performs another pass over the old generation and the (small) delta.

Streaming stays very close to the non-incremental baseline. Its costs consist of the following parts: read original input data on file and compare it with new input data available through streaming so that delta is written (15.3%); process delta (20.8%); merge processed delta with original output (63.9%)—the percentages of costs are given for the rightmost configuration in Fig. 11. Essentially, merging original input and delta dominates the costs of streaming, but those costs are below the costs of processing the new input in non-incremental fashion because the former is a map-side join while the latter is a regular MapReduce computation.

Pipelines In practice, MapReduce jobs are often organized in pipelines or even more complicated networks—remember the use case of crawling in Sec. 2. In such compounds, the benefit of processing deltas as opposed to complete inputs adds up. We consider a simple benchmark that shows the effect of cumulative speedup. That is, four MapReduce jobs are organized in a pipeline, where the first job sorts, as described above, and the subsequent jobs simply copy. Here, we note that a copy job is slightly faster than a sort job (because of the eliminated costs for partitioning for total order), but both kinds of jobs essentially entail zero mapper/reducer costs, which is the worst case for delta-aware computations.

The results are shown in Fig. 12. The chosen pipeline is not sufficient for the “naive” incremental option to outperform the non-incremental option, but the remaining incremental options provide speedup. MapReduce-scenarios in practice often reduce the volume of data along such pipelines. (For instance, the counts of word occurrences require much less volume than the original text.) In these cases, costs for merging go significantly down as well, thereby further improving the speedup.

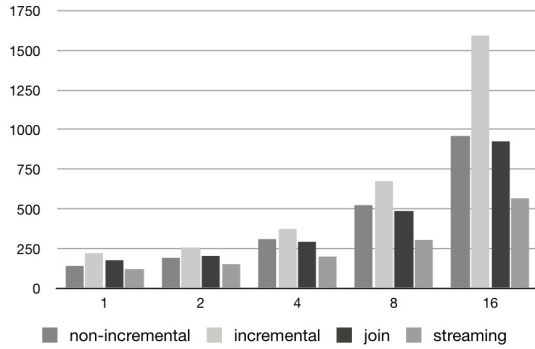


Fig. 12: Sort followed by three copy jobs.

7. Related work

An approach to update *PageRank* computations in the context of changes in the web is introduced by [14]. Similar to our approach, existing results are updated according to computed additions and deletions. However, the approach specifically applies to graph-computations, whereas our approach deals with incremental MapReduce computations in general.

Percolator [15] is Google's new approach in dealing with the dynamic nature of the web. *Percolator* is aimed at updating an existing index that is stored in BigTable [16], Google's high performance proprietary database system. *Percolator* adds trigger-like procedures to BigTable columns, that are triggered whenever data is written to that column in any row. The paper states that *Percolator* requires more resources than MapReduce and only performs well under low crawl rates (i.e., the new input is a small fraction of the entire repository). Our approach uses essentially the same resources than classic MapReduce. We do not understand well enough how to compare our speedups (relative to delta sizes and other factors in our approach) with *Percolator's* scalability (relative to crawl rates).

Twister [17], a distributed in-memory MapReduce runtime, is optimized for iterative MapReduce by several modifications to the original MapReduce model. Iterative jobs are run by a single MapReduce task, to avoid re-loading static data that does not change between iterations. Furthermore, intermediate data is not written to disk, but populated via distributed memory of the worker nodes. CBP, a system for *continuous bulk processing* [18], distinguishes two kinds of iterative computations: several iterations over the same input (e.g., *PageRank*), and iteration because of changed input (e.g., *URLCount*). CPB introduces persistent access to state re-use prior work along reduction. Our approach does not introduce state, which contributes to the simple correctness criterion for MapReduce computations with deltas. Our approach does not specifically address iterative computations, but instead it enables a general source for speedup for MapReduce computations.

Dryad [19], [20] is a data-parallel programming model like MapReduce, which, however, supports more general

DAG structures of dataflow. *Dryad* supports reuse of identical computations already performed on data partitions and incrementality with regard to newly appended input data for which computed results are to be merged with previous results. While the idea of merging previous and new results is similar to deltas, our approach is not restricted to append-only scenarios.

Map-reduce-merge [21] enhances MapReduce to deal with multiple heterogeneous datasets so that regular MapReduce results are merged in an extra phase. The enhanced model can express relational algebra operators and implement several join-algorithms to unite multiple heterogeneous datasets. In contrast, the merge phase in our approach is a problem-independent element of the refined programming model which simply combines two datasets of the same structure.

For our implementation we used Hadoop [3], an open source Java implementation of Google's MapReduce framework [1]. Hadoop's MapReduce-component [22] is built on top of HDFS [23], the *Hadoop Distributed File System* which has been modeled after the *Google File System (GFS)* [24]. Hadoop happens to provide a form of streaming (i.e., *Hadoop Streaming*) for the composition of MapReduce computations [25]. This form of streaming is not directly related to streaming in our sense of delta computation.

MapReduce Online [26] is a modified MapReduce architecture which introduces pipelining between MapReduce jobs as well as tasks within a job. The concept is implemented as a modification of Hadoop. A more general stream-based runtime for cloud computing is *Granules* [27]. It is based on the general concept of *computational tasks*, that can be executed concurrently on multiple machines, and work on abstract datasets. These datasets can be files, streams or (in the future) databases. Computational tasks can be specialized to map and reduce tasks, and they can be composed in directed graphs allowing for iterative architectures. *Granules* uses *NaradaBrokering* [28], an open-source, distributed messaging infrastructure based on the publish/subscribe paradigm, to implement streaming between tasks. We believe that such work on streaming may be helpful in working out streaming deltas in our sense.

Our programming model essentially requires that reduction is based on the algebraic structure of an Abelian group. This requirement has not been set up lightly. Instead, it is based on a detailed analysis of the MapReduce programming model overall [2], and a systematic review of published MapReduce use cases [6].

8. Conclusion

We have described a refinement of MapReduce to deal with incremental computations on the grounds of computing deltas, and merging previous results and deltas possibly throughout pipelines. This refinement comes with a simple correctness criterion, predictable speedup, and it can be

provided without any changes to an existing MapReduce framework. Our development is available online.

There are some interesting directions for future work.

The present paper focuses on the principle speedup and the correctness of the method. A substantial case study would be appreciated to reproduce speedup in a complex scenario. For instance, an existing WebCrawler could be migrated towards MapReduce computations with deltas.

Currently, we do not provide any reusable abstractions for streaming delta. In fact, the described benchmark for streaming TeraByte Sort relies on summation of assumed components of the computation, but we continue working on an experimental implementation.

Our approach to streaming delta and map-side join for merge may call for extra control of task scheduling and file distribution. For instance, results of processing the delta could be stored for alignment with the original result so that map-side join is most efficient.

As the related work discussion revealed, there is a substantial amount of techniques for optimizing compound data-parallel computations. While the art of benchmarking classic MapReduce computations has received considerable attention, it is much harder to compare the different optimizations that often go hand in hand with changes to the programming model. On the one hand, it is clear that our approach provides a relatively general speedup option. On the other hand, it is also clear that other approaches promise more substantial speedup in specific situations. Hence, a much more profound analysis would be helpful.

Modern MapReduce applications work hand in hand with a high performance database system such as BigTable. The fact that developers can influence the locality of data by choosing an appropriate table design, could enable very efficient delta computations. Database systems such as BigTable also offer the possibility to store multiple versions of data using timestamps. This could facilitate delta creation substantially.

Acknowledgment The authors are grateful for C. Litauer and D. Haussmann's support in setting up a MapReduce cluster at the University of Koblenz-Landau for the purpose of benchmarking.

References

- [1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2004, pp. 137–150.
- [2] R. Lämmel, "Google's MapReduce programming model—Revisited," *Science of Computer Programming*, vol. 70, no. 1, pp. 1–30, 2008.
- [3] "Apache Hadoop," <http://hadoop.apache.org/>.
- [4] "Apache Nutch," <http://nutch.apache.org/>.
- [5] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Stanford Digital Library Technologies Project, Tech. Rep., 1998.
- [6] A. Brandt, "Algebraic Analysis of MapReduce Samples," 2010, Bachelor Thesis, University of Koblenz-Landau.
- [7] J. W. Hunt and M. D. McIlroy, "An Algorithm for Differential File Comparison," Bell Laboratories, Tech. Rep., 1976.
- [8] J. Venner, *Pro Hadoop*. Apress, 2009.
- [9] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.
- [10] D. Coutts, R. Leshchinskiy, and D. Stewart, "Stream fusion: from lists to streams to nothing at all," in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*. ACM, 2007, pp. 315–326.
- [11] "Sort Benchmark," web site <http://sortbenchmark.org/>.
- [12] O. O'Malley, "TeraByte Sort on Apache Hadoop," 2008, contribution to [11].
- [13] A. C. Murthy, "Winning a 60 second dash with a yellow elephant," 2009, contribution to [11].
- [14] P. Desikan and N. Pathak, "Incremental PageRank Computation on evolving graphs," in *Special interest tracks and posters of the 14th international conference on World Wide Web*, ser. WWW '05. ACM, 2005, pp. 10–14.
- [15] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10, 2010, pp. 1–15.
- [16] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06, 2006, pp. 205–218.
- [17] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: a runtime for iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC 2010*. ACM, 2010, pp. 810–818.
- [18] D. Logothetis, K. C. Webb, C. Olston, K. Yocum, and B. Reed, "Stateful Bulk Processing for Incremental Analytics," in *SoCC '10 Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 51–62.
- [19] M. Isard, M. Budi, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2007 EuroSys Conference*. ACM, 2007, pp. 59–72.
- [20] L. Popa, M. Budi, Y. Yu, and M. Isard, "DryadInc: Reusing work in large-scale computations," in *HotCloud'09 Proceedings of the 2009 conference on Hot topics in cloud computing*, 2009.
- [21] H. chih Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker Jr., "Map-reduce-merge: simplified relational data processing on large clusters," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2007, pp. 1029–1040.
- [22] "Hadoop MapReduce," <http://hadoop.apache.org/mapreduce/>.
- [23] D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*, The Apache Software Foundation, 2007.
- [24] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. ACM, 2003, pp. 29–43.
- [25] "Hadoop Streaming," <http://hadoop.apache.org/common/docs/r0.15.2/streaming.html>, 2008.
- [26] T. Condie, N. Conway, P. Alvaro, J. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proceedings of the 7th Symposium on Networked Systems Design and Implementation*, ser. NSDI'10. USENIX Association, 2010, pp. 313–328.
- [27] S. Pallickara, J. Ekanayake, and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support, for MapReduce," in *Proceedings of the 2009 IEEE International Conference on Cluster Computing*. IEEE, 2009, pp. 1–10.
- [28] S. Pallickara and G. Fox, "NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids," in *Proceedings of 2003 ACM/IFIP/USENIX International Middleware Conference*. Springer, 2003, pp. 41–61.

Towards Utilizing Remote GPUs for CUDA Program Execution

Xiaonan Ji¹, Spencer Davis², Erikson Hardesty², Xu Liang², Sabuj Saha², and Hai Jiang²

¹Department of Computer Science, Ohio State University, Columbus, Ohio 43210, USA

²Department of Computer Science, Arkansas State University, Jonesboro, Arkansas 72467, USA

Abstract—*The modern CPU has been designed to accelerate serial processing as much as possible. Recently, GPUs have been exploited to solve large parallelizable problems. As fast as a GPU is for general purpose massively parallel computing, some problems require an even larger scale of parallelism and pipelining. However, it has been difficult to scale algorithms beyond a local computer and distribute workloads among multiple computers housing GPUs. This paper proposes a Remote Kernel Launch (RKL) approach to transfer the kernel parts from a local machine to remote GPU servers. A lexical analyzer is used to identify and extract the kernels from local programs. The extracted kernel can then be distributed and executed on remote GPUs. A dynamic mapping scheme is explored to balance workloads among nodes. This approach allows a program to run optimally on a range of hardware configurations by eliminating the need to program for a specific environment. The experimental results demonstrate the effectiveness of RKL.*

Keywords: Remote Procedural Call, Graphics Processing Unit, CUDA

1. Introduction

Even in the current technology era, many computational intensive problems remain difficult to solve with a single monolithic CPU. As Moore's law gives us more and more transistors, the strategy with CPUs is to make the workload (one computer thread) run as fast as possible through the use of large caches, instruction/data prefetch and speculative execution. However, the CPU's performance is limited by the communication bandwidth, which lead to the consideration of parallelism. Multiple CPU execution paths, or cores, are being used to process threads of execution. Two to eight cores are commercially available now, and it is predicted that hundreds of cores will be achieved in the future.

In the meantime, another hardware approach is being used to exploit this parallelism. A graphics processing unit or GPU is a specialized circuit that is capable of rendering graphics much more quickly than a central processor [4]. A single modern GPU already contains more than 500 computational cores. GPUs process blocks of data in parallel on a much larger scale than a CPU. Because of this, GPUs have begun making computational in roads against the CPU. This concept turns the massive floating-point computational power of a modern graphics accelerator's shader pipeline

into general-purpose computing power, as opposed to being hard wired solely to do graphical operations.

For very large problems It remains difficult to scale execution to remote GPUs as compared to the more typical CPU based supercomputer. This combined with the lower level programming required to take advantage of GPUs has slowed adoption in many areas. While benchmarks show massive performance gains with GPUs, the limited tools available to exploit this raw performance as compared to CPUs has hindered its practical uses.

This paper makes the following contributions:

- The RKL scheme is deployed to utilize remote GPU computing resources in an efficient way. A lexical analyzer written in Flex is adopted to identify and extract the kernel parts from local programs.
- A dynamic mapping scheme is developed to distribute the kernel parts onto remote GPUs. The mapping scheme maintains efficiency of solving a single task as ensuring a balanced workload among nodes.
- The tradeoff between the overhead for the remote call and the acceleration of computing with different problem sizes is considered.

The remainder of this paper is organized as follows: Section 2 gives an overview of the background of related packages. Section 3 is the design and development of RKL. Section 4 is about feature analysis and experiment results. Section 5 gives the related work. Finally, our conclusion and future work are described.

2. Background

2.1 GPU Computing

Many of today's intensive programming problems have become a critical part of a variety of fields, such as data mining, machine learning, evolutionary computation, life science, image processing, statistic, etc. For these complex problems which require massive vector/matrix operations, GPUs work extremely well because of their high memory bandwidth and massive parallel computation power. In certain applications requiring massive vector/matrix operations, a GPU can yield several orders of magnitude higher performance than a conventional CPU. In addition to raw performance GPUs have other advantages. GPUs have much higher performance per watt than CPUs. When scaled to large supercomputers this performance per watt gap can make a large difference in the cost to operate. In addition,

GPUs can be purchased for relatively small cost. Their performance per cost far exceeds the modern CPU which enables more practical supercomputing for businesses and research.

With the flood of data produced by modern scientific instruments, fast analysis or computing of very large volumes of data is now of paramount importance in many fields. These computationally intensive problems require an even larger scale of parallelism and pipelining. Software developers are more and more inclined to take advantage of GPUs in order to achieve the desirable level of performance.

2.2 CUDA

The RKL system is based on the CUDA (Compute Unified Device Architecture) programming model released by NVIDIA [5], [8]. NVIDIA's CUDA platform is the most widely adopted programming model for GPU computing. CUDA enables GPU hardware to be exploited for higher performance [9]. CUDA allows specified functions from a normal C program to run on the GPU's stream processors. With CUDA, C programs are capable of taking advantage of a GPU's strengths, while still making use of the CPU where appropriate. Programmers can also use CUDA to target multiple GPUs to accelerate computation further. This involves decomposing tasks even further to process them simultaneously on multiple GPUs. CUDA's high-performance scalable computing architecture solves complex parallel problems hundreds of times faster than traditional CPU-based architectures. CUDA does not provide an interface to distribute compute kernels to remote systems. Few tools currently exist to address this issue.

2.3 LEX/FLEX

Lex is a program generator designed for lexical processing of character input streams. It helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

Flex is a free version of Lex distributed by GNU. The RKL system exploits Flex to implement its lexical analyzer, which identifies various elements of the CUDA C source code.

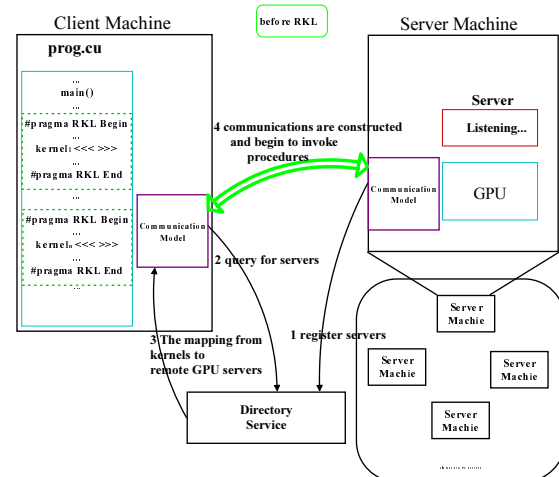


Fig. 1: System layout of RKL

3. Design and Development of RKL

3.1 System Layout

The Remote Kernel Launch, RKL system consists of three parts: client machines, directory service [7], [3] and a cluster of GPU servers. A client machine sends a request to the directory service when a RKL is needed in a program. The directory service, which is the software system that stores, organizes and provides access to resources in a directory, records the status of every GPU server in the cluster and maps kernels from the client side to GPU server. A cluster of GPU servers is composed of multiple computers housing one or more GPUs. Each GPU server registers their information on the directory service and notifies the service when it acquires and completes a task. An advantage with this system is that the client machines and programs are abstracted from the execution hardware. The client need not know anything about the GPU server in the cluster. All decisions are made by the directory service [6]. The system status before the Remote Kernel Launch is shown in Fig. 1

One critical issue to tackle in this design lies in the measurement of kernel complexity and GPU capacity. Determining the kernel complexity is essential to find the GPU server with the best compatibility. The complexity of one kernel can be directly reflected by the number of its threads, which can be calculated from the code segment `<<< griddim,blockdim>>>`, where `griddim * blockDim` is the number of threads that needs to be generated for a kernel. To estimate the compatibility of one GPU server, we need to derive its max thread number from its version information which is registered and recorded in the directory service. The max thread number is one GPU metric to determine the best fit for a kernel. This could be extended to other metrics to determine optimal compatibility.

For dynamic workloads, daemons can be installed on both servers and clients. It is possible that the program running on

one client machine can possess one or multiple kernels. After taking the tradeoff between the overhead of the remote call and the acceleration on operation speed into consideration, the complexity of each kernel and the capacity of local GPU will be analyzed to determine whether the local support is enough and whether RKL is needed. If the local GPU can handle one kernel with efficiency, then the kernel will be executed locally. If the local GPU is busy then RKL will be applied. A flexible mapping scheme is explored to support RKL in an effective way.

3.2 Lexical Analyzer

The primary task of the lexical analyzer is to identify and extract the CUDA parts of the source program as shown in Fig. 2 for a typical layout. First, the CUDA related code is removed from the client program source. The remainder of the client-side program is traditional C code which is runnable in a local machine without CUDA-supported GPUs. In place of the original CUDA code, a function named "remote_kernel_launch" will be invoked as shown in Fig. 3.

The extracted CUDA code is not a complete program and must be supplemented with additional code. This new intermediate CUDA program is only source code and does not contain data associated with the original client program's state prior to the kernel launch. Therefore, the intermediate CUDA program has to handle the transfer of data before it can proceed to computation tasks as shown in Fig. 4.

The RKL function in the client first creates a connection with the directory service and requests a capable server. Once it connects with the assigned server, it sends the now complete intermediate CUDA program. When the program arrives and is then invoked at the server end, an acknowl-

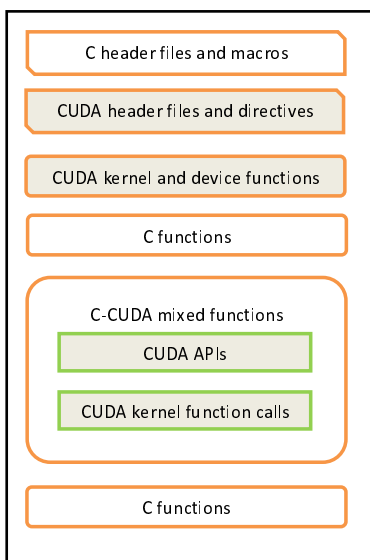


Fig. 2: Original CUDA program

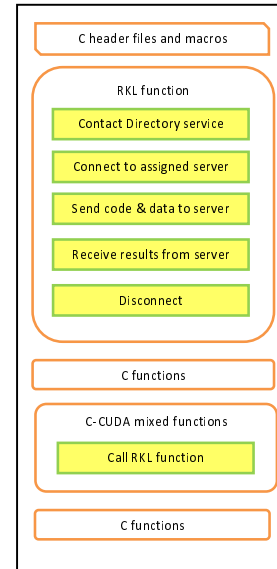


Fig. 3: Translated client program (C code with socket)

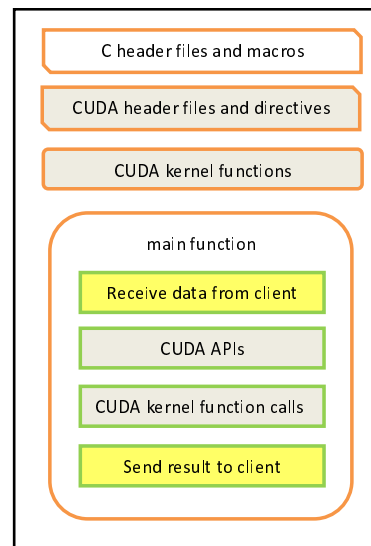


Fig. 4: Translated server program (CUDA code with socket)

gement is sent out to ask the client side program for data. After the data has been received the computation task can be completed. The server then transmits the results back to the client. When the "RKL" function returns, the client side program proceeds.

3.3 Mapping Scheme

RKL is implemented for intensive kernels that need stronger support from remote GPU servers. The mapping scheme of RKL intends to find the GPU server with the highest compatibility for every kernel, that is, one GPU server that can not only efficiently complete the task but also

minimize the possible idle waste. Assume two GPU servers can support up to 15,000 and 30,000 threads, respectively. Running a kernel with 13,000 threads on the GPU with 30,000 threads will lead to more idle waste than running the kernel on the 15,000 one, especially when the GPU with 15,000 threads can already provide sufficient support. The situation will be worse when a second kernel with 28,000 threads comes after the first kernel, and the GPU with 30,000 threads is occupied by the first kernel at the same time. In this sense, the second kernel will have to run on the GPU with 15,000 threads where its performance will drop. This situation can be avoided if we schedule the kernel with 13,000 threads to the GPU with 15,000 capacity, and the kernel with 28,000 threads to the GPU with 30,000 threads capacity. In the proposed mapping scheme, kernels are assigned to several classes according to their complexity indicated by the number of threads. Each class will occupy a continuous region on the complexity axis such as (10,000, 16,000), (16,000, 23,000), (23,000, 30,000). Similarly, GPUs are assigned to several classes based on their thread capacity and also occupy continuous regions on the max thread number axis. Every kernel class is mapped to one GPU class for better compatibility. Such mapping scenario is shown in Fig. 5.

3.4 Remote Kernel Launch (RKL)

Fig. 6 illustrates the event sequence when RKL occurs. For one program file on the local machine, a lexical analyzer is applied to identify and split the intensive kernel parts in the source code. In this approach, with the lexical analyzer, each kernel marked by #pragma will be extracted out and generates a new CUDA file which will be sent to the remote GPU server mapped by the directory service. The rest of the source code after the extraction will be completed by adding some instructions like socket(), send(), receive(), which in turn control the communication with the server while sending arguments and receiving results. After

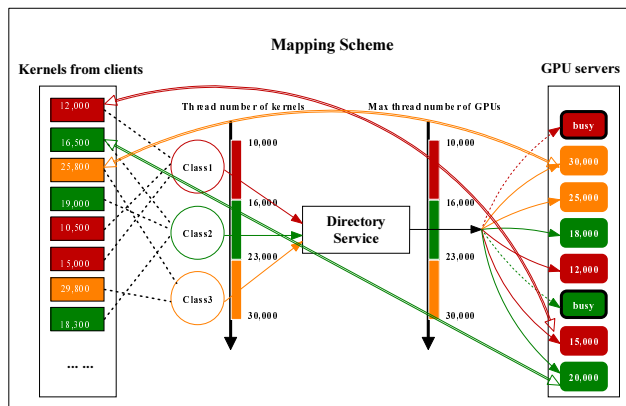


Fig. 5: Mapping between kernel requests and GPU server capacity

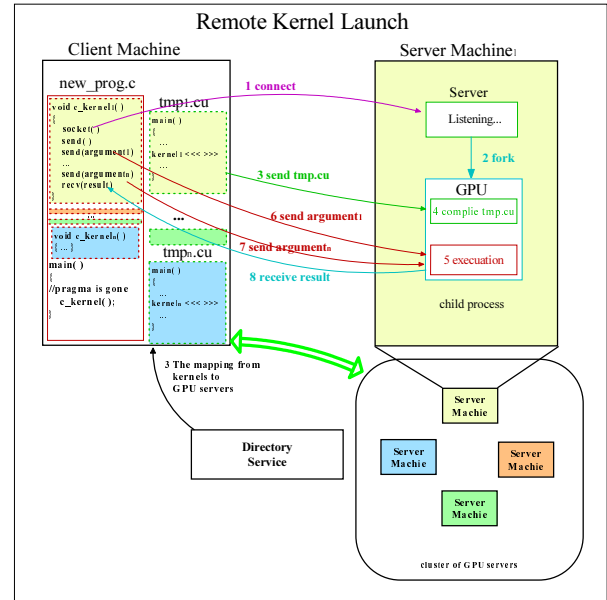


Fig. 6: Event sequence in Remote Kernel Launch

a connection is established between the client machine and the server machine, the remote GPU server then forks a child process to compile the CUDA file and execute the new program made by the compilation. Within the CUDA file a new connection should be constructed to take responsibility of the transmission of arguments and results. The GPU server will request for the arguments from the local client machine, and the local client machine will send the requested arguments and receive the final results from the remote GPU server. In this sense, the intensive tasks can be distributed to a cluster of GPU servers effectively and efficiently.

4. Experimental Results

In order to facilitate the communication between the client and server machines, the original CUDA file must be parsed and analyzed to generate the client.c and server.cu files. Fig. 7 contains an example CUDA program before it is parsed by the lexical analyzer. Two sections in this example must be extracted out, and placed into the server.cu file. The first is indicated by the __global__ specifier, which identifies a CUDA kernel function. The second section is inside the main function, indicated by the #pragma tags. The use of the #pragma tags allows the program to be run without modification in an environment without RKL capabilities. The remaining portions are used to generate the client.c file.

Fig. 8 shows the generated client.c file. This file contains all of the non-CUDA portions of the original CUDA file, as well as an added function that will facilitate communications between the client, directory service, and the assigned server. A call to this function is inserted into the main function in place of the #pragma specified code. The generated server.cu

```

#include <stdio.h>
#include <sys/time.h>
#define SIZE 2048
int A[SIZE], B[SIZE], C[SIZE];

__global__ void vecAdd(int* A, int* B, int* C) /*CUDA kernel
function*/
{
    unsigned int i = threadIdx.x+blockIdx.x* blockDim.x;
    C[i]=A[i]+B[i];
}

int main(int argc, char *argv[])
{
    int i, N=SIZE;
    srand((unsigned)time(NULL));
    int memsize = SIZE*sizeof(int);
    for (i=0; i<N; i++){
        A[i]=(int)((double)rand()/(RAND_MAX+1.0)*5);
        B[i]=(int)((double)rand()/(RAND_MAX+1.0)*5);
    }

    #pragma RKL Begin
    int *devPtrA, *devPtrB, *devPtrC;
    cudaMalloc((void **)&devPtrA, memsize);
    cudaMalloc((void **)&devPtrB, memsize);
    cudaMalloc((void **)&devPtrC, memsize);
    cudaMemcpy(devPtrA, A, memsize, cudaMemcpyHostToDevice);
    cudaMemcpy(devPtrB, B, memsize, cudaMemcpyHostToDevice);
    /*CUDA kernel function call*/
    vecAdd<<<64, 32>>>(devPtrA, devPtrB, devPtrC);
    cudaMemcpy(C, devPtrC, memsize, cudaMemcpyDeviceToHost);
    cudaFree(devPtrA);
    cudaFree(devPtrB);
    cudaFree(devPtrC);
    #pragma RKL End

    for(i=0; i<SIZE; i++) printf("C[%d]=%d\n", i, C[i]);
    return 0;
}

```

Fig. 7: The original vector addition CUDA code (vecAdd.cu)

file can be seen in Fig. 9. This file contains the extracted CUDA portions, as well as socket programming to receive arguments and send back results.

Experimental results using the aforementioned vector addition kernel showed reasonable overhead given a kernel of sufficiently high computational complexity. In practice, the time from the start of the RKL to the actual execution of the kernel on the GPU server was approximately a second. Larger kernels would have more overhead due to increase compilation and transfer time. However, most GPU kernels are relatively short in length and can be compiled very quickly. While, the data needed to compute the kernel would also have an effect on the RKL overhead, this data transfer would be needed if using a traditional CPU based distributed system as well.

5. Related Work

Recent designs and approaches have made GPUs more programmable and useful for tasks beyond graphics, due to GPUs' high computing capacity. While CUDA is a big step towards GPU programming, it requires significant rewriting and restructuring of programs. PGI is introducing the Accelerator Programming Model for Fortran and C with PGI Release 9.0 [10]. The Accelerator Programming Model uses directives and compiler analysis to compile natural Fortran and C for the GPU, which often allows users to maintain a single source version. This model is designed to be forward-looking as well, in order to accommodate

```

#include <stdio.h>
#include <sys/time.h>
#define SIZE 2048
int A[SIZE], B[SIZE], C[SIZE];

/* Newly inserted function for communications */
void vecAdd_RKL(int* A, int* B, int* C)
{
    ...
    /* Contact Directory Service by socket */
    ...
    /* Connect to server.cu on the assigned GPU server */
    /* Send kernel arguments array A and B to server.cu */
    ...
    /* Receive results array C from server.cu */
    ...
    /* Disconnect socket communication with the server.cu */
    ...
}

int main(int argc, char *argv[])
{
    int i, N=SIZE;
    srand((unsigned)time(NULL));
    int memsize = SIZE*sizeof(int);
    for (i=0; i<N; i++){
        A[i]=(int)((double)rand()/(RAND_MAX+1.0)*5);
        B[i]=(int)((double)rand()/(RAND_MAX+1.0)*5);
    }

    /* newly inserted function call */
    vecAdd_RKL(A, B, C);

    for(i=0; i<SIZE; i++) printf("C[%d]=%d\n", i, C[i]);
    return 0;
}

```

Fig. 8: The generated client-side code (client.c)

other accelerators that may come in the future, preserving the software development investment.

HMPP is a language extension for hardware accelerators for C, Fortran, and C++ for multi-thread programming [2]. HMPP targets to be applied to CUDA for NVIDIA GPU with its compiler directives which has no code modifications (just comments if not recognised by the compiler) and mostly hardware independent. HMPP can address different targets and strategies. The difference between HMPP and RKL is that the former focuses on local execution whereas the latter is for remote kernel function launch.

RPC (Remote Procedure Call) has been in computer world for a long time [1]. However, RPC deals with portable function execution on CPUs. RKL extends CUDA programs' working platforms from local machines to distributed systems or even Clouds.

6. Conclusion and Future Work

This paper describes the deployment of Remote Kernel Launch to dispatch large and highly parallel computations across networked GPU servers through the cooperation of both a client side and a server side. A lexical analyzer is applied to identify and extract CUDA kernels. With a dynamic mapping scheme, these kernels can be executed on the GPU servers with high compatibility so that high overall efficiency can be achieved. RKL enables high-efficiency computing for intensive problems in a wide range of areas with the maximized utilization of available resources. The RKL method described requires only minor changes to existing CUDA

```

#include <stdio.h>
#include <sys/time.h>
#define SIZE 2048
int A[SIZE], B[SIZE], C[SIZE];

__global__ void vecAdd(int* A, int* B, int* C) /*CUDA kernel
function*/
{
    unsigned int i = threadIdx.x+blockIdx.x* blockDim.x;
    C[i]=A[i]+B[i];
}

int main(int argc, char *argv[])
{
    int N=SIZE;
    int memsize = SIZE*sizeof(int);
    ...
    /* Receive kernel argument A and B from client.c on client machine */
    ...

    int *devPtrA, *devPtrB, *devPtrC;
    cudaMalloc((void**)&devPtrA, memsize);
    cudaMalloc((void**)&devPtrB, memsize);
    cudaMalloc((void**)&devPtrC, memsize);
    cudaMemcpy(devPtrA, A, memsize, cudaMemcpyHostToDevice);
    cudaMemcpy(devPtrB, B, memsize, cudaMemcpyHostToDevice);
    /*CUDA kernel function call*/
    vecAdd<<<64, 32>>>(devPtrA, devPtrB, devPtrC);
    cudaMemcpy(C, devPtrC, memsize, cudaMemcpyDeviceToHost);
    cudaFree(devPtrA);
    cudaFree(devPtrB);
    cudaFree(devPtrC);

    /* Send results in C back to client.c on the client machine */
    ...
    /* Disconnect the socket channel */
    return 0;
}

```

Fig. 9: The generated server-side code (server.cu)

programs in order to operate in a scalable heterogeneous environment. Additionally, the changes needed to the CUDA program are backwards compatible without modification needed to run in a non-RKL environment. Experiments have confirmed the feasibility of this approach demonstrating that RKL has minimal computational overhead.

There are several opportunities for future work. The mapping scheme could be extended to include more metrics in the process of deciding the GPU server with the best fit for the particular GPU task. Other work could include implementing the dynamic mapping scheme with publish/subscribe and peer-to-peer style systems. Additionally work could be done to implement the RKL system in OpenCL. Eventually, RKL will be ported to Cloud computing systems.

References

- [1] Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 1984.
- [2] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Proceedings of Workshop on General Purpose Processing on Graphics Processing Units*, 2007.
- [3] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith, and Steven Tuecke. A directory service for configuring high-performance distributed computations. In *Proceedings of The Sixth IEEE International Symposium on High Performance Distributed Computing*, 1997.
- [4] Wen-mei Hwu. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann, 2011.
- [5] David Kirk and Wen-mei Hwu. *Programming Massively Parallel Processors, A Hands-on Approach*. Morgan Kaufmann, 2010.
- [6] R. Rajkumar, M. Gagliardi, and Sha Lui. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation. In *Proceedings of Real-time Technology and Applications Symposium*, 1995.
- [7] J. Reynolds and C. Weider. Executive introduction to directory services using the x.500 protocol, rfc 1308. Technical report, 1992.
- [8] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General -Purpose GPU Programming*. Addison-Wesley Professional, 2010.
- [9] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with gpu accelerators. In *Proceedings of IEEE International Parallel & Distributed Processing Symposium*, 2010.
- [10] Michael Walfe. Optimizing data movement in the pgi accelerator programming model, February 2011.

Power Saving Mechanism for Multi-cluster Resource Manager with Dynamic Loading Prediction Scheduling Algorithm

Chang-Hsing Wu¹, Yi-Lun Pan¹

¹National Center for High-Performance Computing, Hsinchu, Taiwan
e-mail : hsing@nchc.org.tw, serenapan@nchc.org.tw

Abstract – The “Green Computing,” is especially important and timely. As 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. A high performance-computing cluster is a set of computers gathered together and connected with the intent of merging their processing power towards the same goal. In this paper we developed a multi-cluster resource manager with proposed scheduling algorithm to solve this problem. The multi-cluster resource manager can control and balance loading of various heterogeneous clusters more efficiently. To echo today’s energy saving issues, we also proposed a power saving mechanism. Furthermore, this algorithm also considers several properties of the multi-cluster system, including heterogeneous, dynamic adaptation and the dependent relationship of jobs. And, the power saving mechanism can wake up these idle machines when computing power is requested and power down when the job is done.

Keywords: Job Scheduling, Multi-Cluster, Meta-scheduler, Scheduler, Power Saving

1 Introduction

In the last few years, the trends in parallel processing system design and deployment has moved away from single isolated powerful supercomputer to cooperative networked distributed systems such as commodity-based cluster computing and distributed computing systems [1]. Cluster computing systems can be single cluster system or multi-cluster system [2] [3]. A single cluster system is formed from a set of independent workstations that are interconnected by a local area network (LAN). Multi-cluster system is formed from a set of independent clusters interconnected by a wide-area network (WAN) [4] in Fig. 1.

Recently, multi-cluster system is gaining more importance in practice and a wide variety of applications are being hosted on such systems as well. Also, it has been shown that parallel applications that have been written for homogeneous single cluster systems do not run efficiently on multi-cluster system. Hence, we will further focus on multi-cluster system.

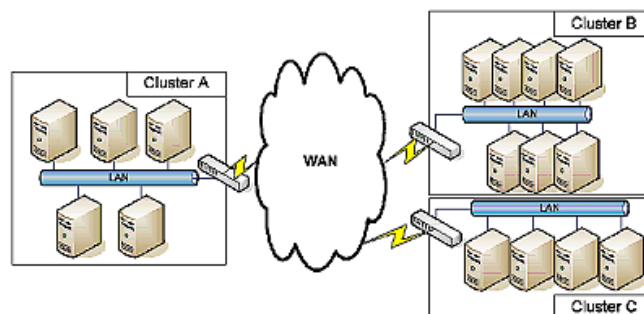


Fig. 1. The multi-cluster system as a group of clusters interconnected through a wan.

In order for multi-cluster system to work efficiently, we designed and developed a multi-cluster resource manager (MCRM), which is a meta-scheduler for multi-cluster system. The multi-cluster resource manager then drives the local scheduler to enable the cluster computing. The MCRM drives the local scheduler of clusters of multi-cluster system via network socket and secure shell (SSH). It has complete meta-scheduler framework such as request manager, job manager, dispatch manager and scheduling module...etc. The MCRM reduces a heavy burden on clusters interconnect and complex framework of multi-cluster system.

We also designed a dynamic loading prediction scheduling algorithm and implemented it on scheduling module. It is developed according to each different required job criteria, and the scheduling algorithm uses the dynamic loading prediction and adaptive resource allocation functions to meet users’ requirements. The major task of the proposed multi-cluster resource manager is to dynamically identify and characterize the available resources, correctly monitor the queue status of local scheduler. Finally, the presented scheduling algorithm helps to select and allocate the most appropriate resources for each given job. The aim of the presented scheduling algorithm is to minimize the total time of deliver for the individual users’ requirements and criteria. Besides, the “Green Computing,” is especially important and timely. As 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 an approach to reduce energy utilization in local cluster. 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.

According to the above scenario, those issues encourage the motivation of our research and development. Specifically, the design and implementation of the proposed MCRM includes the dynamic loading prediction and adaptive resource selection functions. Moreover, the scheduling algorithm takes into consideration not only the general features of the multi-cluster resource manager but also the "dynamic" feature, i.e. it monitors the status of each local queue and provides resources according to users' criteria dynamically. Therefore, the proposed MCRM can select resource efficiently and automatically. Further, the effective job scheduling algorithm also can improve the performance and integrate the resources to supply remote user efficiency in the heterogeneous and dynamic multi-cluster system.

The rest of the paper is organized as follows. Section 2 presents a formal definition of the multi-cluster computing and existing resource manager and scheduling algorithm. In Sections 3, we propose a new scheduling algorithm, power saving mechanism and describe the architecture of MCRM that we developed. In Section 4, the performance comparison of algorithms is presented. The conclusion and future directions are presented in Section 5.

2 State of the Art

2.1 Multi-Cluster System

The using of multi-cluster system can be a cheap, flexible and adaptable alternative to reduce applications' execution time. Nevertheless a multi-cluster system is a complex environment whose heterogeneity challenges the collaborative execution of applications and their efficient speedup. The complexity grows when scattered clusters interconnected through Internet form a multi-cluster system.

The work presented in this thesis targets to reduce the execution time of several applications written for a single cluster, using a multi-cluster system. In order to achieve this goal, we designed and developed a middleware – multi-cluster resource manager (MCRM) that allows a collection of clusters to form a multi-cluster system. The architecture of the MCRM efficiently to organize cluster's resources and produces a methodology to guide in the process of achieving, using multi-cluster system, an application speedup with a certain level of efficiency.

This paper focuses on our designed architecture of the MCRM and scheduling of dependent jobs, which means have a reciprocal effect, or correlation between each other. Because of the scheduling, it can help multi-cluster system to increase and integrate the utilization of cluster computing

resources. Therefore, scheduling can improve the performance in the multi-cluster system.

2.2 Existing Resource Manager for Multi-Cluster System and Scheduling Algorithm

A general architecture of scheduling for the multi-cluster resource manager (MCRM) which is defined as the process of making scheduling decisions involving resources over multiple administrative clusters. There are three important features of the MCRM, which are resource monitoring, resource selection, and job execution. As we know, a lot of researches of resource broker or meta-scheduler are on going to provide access of resources for different applications, such as Condor – G, EDG Resource Broker, AppLes, and so on [5], [6]. The above resource brokers also can provide the capability of computing resource monitoring and resource selection. Nevertheless, those researches do not deal with monitoring the information of dynamic queuing, neither to make precise and effective scheduling policy. The dynamic job scheduling is crucial and fundamental issue in multi-cluster system. The purpose of job scheduling is to find the dynamic and optimal method of resource allocation. Most researchers are applying the traditional job scheduling strategies to allocate computing resources statically, such as list heuristic and the listing scheduling (LS) [7]. The above algorithms focus on the allocation of machines statically. However, our research focuses on dynamic job scheduling for each job from users' requirements and criteria.

Definition 1: The list heuristic scheduling algorithms have three variants - First-Come-First-Serve (FCFS) - the scheduler starts the jobs in the order of their arrival time; Random - the next job to be scheduled is randomly selected among all jobs. No jobs are preferred; Backfilling [8] - the scheduling strategy is out-of-order of FCFS scheduling that tries to prevent the unnecessary idle time. Actually, there are two kinds of backfill. One is EASY-backfilling, and the other is conservative-backfilling.

Furthermore, most of the researches assume jobs are executed independently and statically [4], [9], [10], [11]. In fact, these assumptions are not appropriate in multi-cluster system, since these jobs are always dependent and dynamic. In this research, the proposed algorithm is designed for scheduling the dependent jobs dynamically.

3 Proposed Scheduling Algorithm of MCRM

3.1 Research Objective

The multi-cluster system is formed from a set of independent clusters interconnected by a wide-area network. Each cluster of multi-cluster system has independent local

scheduler. User can submit job with local scheduler. So, the heterogeneous and dynamic properties of cluster status are considered when designing the algorithm of job scheduling. Therefore, the proposed algorithm can make job scheduling to achieve minimum makespan (defined in *Definition 2*), which is to minimize the total time of delivery procedure for the individual users' requirements and criteria. That is the main contribution of this work to present the developed MCRM with job scheduling algorithm. The MCRM can provide the faultless mechanism such that once the users specify user's requirement of resources. Finally, the MCRM will allocate the most appropriate computing cluster to carry out the execution of the application.

Definition 2: The completion time is defined as the time from the job being assigned to one machine until the time the job is finished. The complete time is also called makespan time.

3.2 Model and Architecture

To simplify multi-cluster system, each distributed cluster computing resource can be connected by high-speed network. There is one important component of middleware – MCRM that plays essential role in such an environment. The responsibilities of the MCRM are to monitor the cluster computing resources status, store the information, and satisfy the users' requirements of computing resources. Therefore, the dynamic loading prediction job scheduling algorithm utilizes available computational resources fairly and efficiently in multi-cluster system.

This proposed algorithm is designed for the MCRM of multi-cluster system and NCHC Resource Broker of grid computing [12]. The purpose of the proposed algorithm is to help improve the performance of job scheduling. As a matter of fact, the proposed scheduling algorithm can preserve the good scheduling sequence of optimal or near-optimal solution, which generated the best host candidate or better host candidates. And then, the presented scheduling module can get rid of the unfit scheduling sequence in the searching process for the scheduling problems.

We used gnu gcc, wxWidgets library [13] and ssh2 library [14] to implement the MCRM for Linux cluster system. The MCRM provided controls over batch jobs and distributed compute clusters. It control clusters via local scheduler such as Torque [15], OpenPBS [16], and Moab [17]. The MCRM architecture is designed as shown in the following Fig. 2.

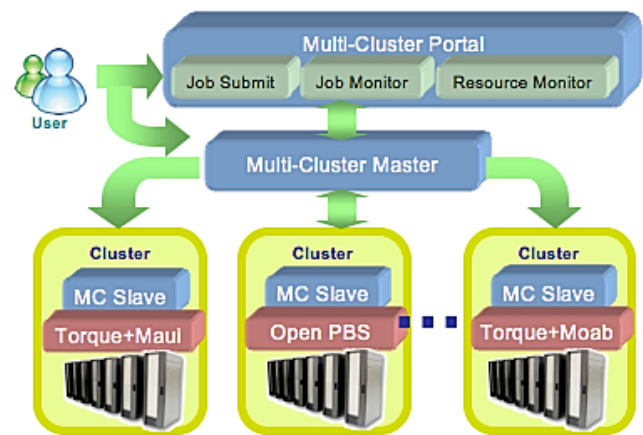


Fig. 2 Multi-Cluster Resource Manager Architecture

The users can submit jobs with using the MCRM directly by terminal or through Multi-Cluster Portal as shown in Fig.2. There are three layers in the system architecture. The first, the Multi-Cluster Portal serves as an interface to the users. It consists of three portlets. Three portlets are Resource Monitor Portlet, Job Monitor Portlet and Job Submit Portlet. Therefore, the jobs are submitted via Job Submit Portlet, which in turn passes the jobs to the MCRM to drive the backend resources. The resource as well as job status are displayed back to the portal for users via either Job Monitor Portlet. The Job Monitor Portlet accesses the job information from the Multi-Cluster Master (MC Master) which maintains the updated information about jobs. Another feature of Job Monitor Portlet is the control over jobs, such as modify job, delete job and resubmit job...etc.

The Resource Monitor Portlet queries the status of resources from the MC Master and displays the results onto the Multi-Cluster Portal. Thus, the users' knowledge about the status of resources is kept up to date. The status content of resources is the loading of clusters, such as CPU availability, number of CPU and number of jobs ...etc.

Secondly the Multi-Cluster Master (MC Master) is the core of the system architecture. It takes the requirements from user's jobs, and then compares with resource information, which is provided by the Multi-Cluster Slave (MC Slave). Therefore, the MC Master can select the most appropriate resources automatically to meet the requirements from jobs. It will further move job's requirement of files (including input files and execution files) to cluster by sftp and submit jobs to local schedulers for processing. The MC Master will continue to monitor until the job is completed. Finally, the output files of job will automatically be moved to the server, which the MC Master is executed.

The last layer, the MC Slave is executed on each cluster of the multi-cluster system. It is used to pass control command from the MC Master to local scheduler. It is also used to collect the dynamic information of local computing resources periodically and update the status of the resources to MC Master. The dynamical information of local

resources, in XML format, is queried from local scheduler, Ganglia, Network Weather Service (NWS), and so on by Perl script. If the administrator wants to control different local scheduler or collects more information form other programs, he only needs to modify the Perl script.

In order to handle the related processes of job submitting, the MCRM adopts the Scheduling Module to find the appropriate scheduling sequence, and then dispatches jobs to the local schedulers by Dispatch Manager Module. The most important part is the core of the MCRM in Fig. 3. The major characteristic of Scheduling Module is the presented scheduling policy, Dynamic Loading Prediction Scheduling (DLPS) algorithm, which provides dynamic loading prediction and adaptive resource allocation functions. The scheduling policy will be described in later section.

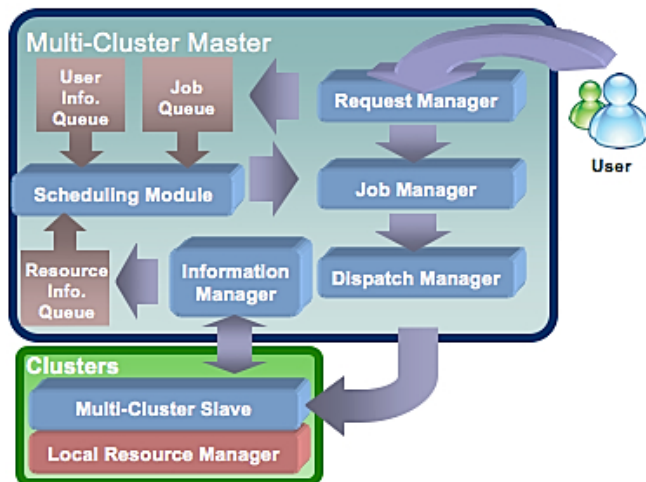


Fig. 3 Multi-Cluster Resource Manager Architecture

3.3 Multi-Cluster Resource Manager Architecture

We will explain the whole architecture of the MCRM, and functions of every component, in this section. As the Fig.3 shows, on receiving a job request via Linux socket protocol from users, the Request Manager will manage the job submission with proper job attributes. The Request Manager will convert the job's requirements into job object and put the object into the Job Queue. On the other hand, the user maintains the control over the jobs submitted through the Request Manager.

Following the Request Manager, the Job Manager invokes the Scheduling Module to generate a resource list based on the criteria posted by the job and the resource status. The Resource Information Queue provides the resource status for the Scheduling Module. The Information Manager updates the object of cluster resource status in the Resource Information Queue periodically. With the suggestion from the Scheduling Module, the job is then dispatched. The Dispatch Manager also provides the ability

to reschedule jobs and report pending status back to the Request Manager.

The Scheduling Module architecture of general algorithm in MCRM has only one function, which is the Job Priority Policy. But, the architecture of Scheduling Module with DLPS has shown in Fig. 4. There are three main functions, namely the Job Priority Policy, the Resource Selection, and the Dynamic Resource Selection. The Job Priority Policy is responsible for initializing the selection process with existing policy such as DLPS, FCFS, Round-Robin, Backfill, Small Job First, Big Job First, and so on. The Resource Selection provides resource recommendation based static information such as hardware specification, specific software, and the High-Performance Linpack Benchmark results.

The Dynamic Resource Selection issues suggestions based on dynamic information such as the users' application requirement, network status as well as work load of individual machines. With the combined efforts of the three components, the Scheduling Module provides the features of the automatic scheduling and re-scheduling mechanism. The automatic scheduling chooses the most appropriate resource candidate followed by the second best choice and so on, while the re-scheduling provides the service to compensate the miss-selection of resource by the users. Once the Scheduling Module provides the best selection of resource, the process is passed to the Job Manager and the Dispatch Manager that drive MC Slave to submit, control and monitor the resource.

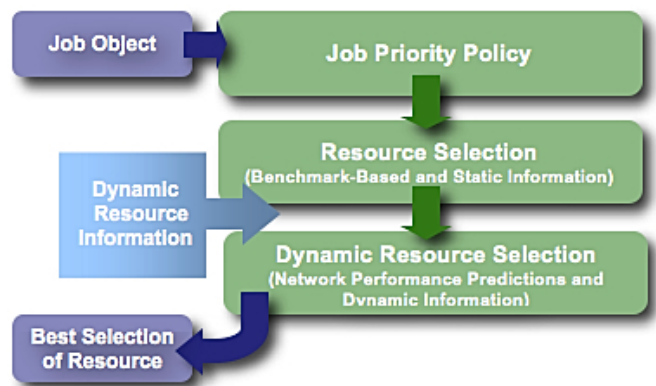


Fig.4 Scheduling Module with Dynamic Loading Prediction Scheduling Algorithm

3.4 Scheduling Policy - Dynamic Loading Prediction Scheduling Algorithm (DLPS)

The proposed job scheduling algorithm of scheduling policy is called Dynamic Loading Prediction Scheduling (DLPS) in the MCRM. The objective function of DLPS is achieving the minimized makespan. Thus, we designed the following equation to describe it, as in (1) :

$$M^* = \text{Min}[\max(d_k) - \min(s_k)] \quad (1)$$

Definition 3: The M^* means the minimized makespan. In order to predict precisely, the equation uses d_k and s_k . d_k is the maximum *job ending time* of the k th job, which means the end time of job completed. And s_k is the minimum *job submitting time* of the k th job, which means the time stamp when users submit the k th job.

There are some inputs, such as job script, job template, and the form of the Multi-Cluster Portal from users' requirements. Therefore, the operations of DLPS are to select, schedule, reschedule, and submit jobs to the most appropriate resources.

The logical flow chart of the DLPS is illustrated as in the Fig. 5. First, the DLPS retrieves the resource information from Information Manager, and then filters out unsuitable resources with the adaptive resource allocation function. After the adaptive resource allocation function, DLPS compares free nodes with required nodes. If current free nodes are enough, DLPS will give higher *weight* (defined at **Definition 4**). Otherwise, the following step enters the dynamic loading prediction function with *EstBacklog* and minimum *Job Expansion Factor* (defined at **Definition 5** and **Definition 6**) methods to predict which computing resources respond and execute job quickly, and then calculates weight. The DLPS finally ranks all available resources and selects the most appropriate resources to dispatch job.

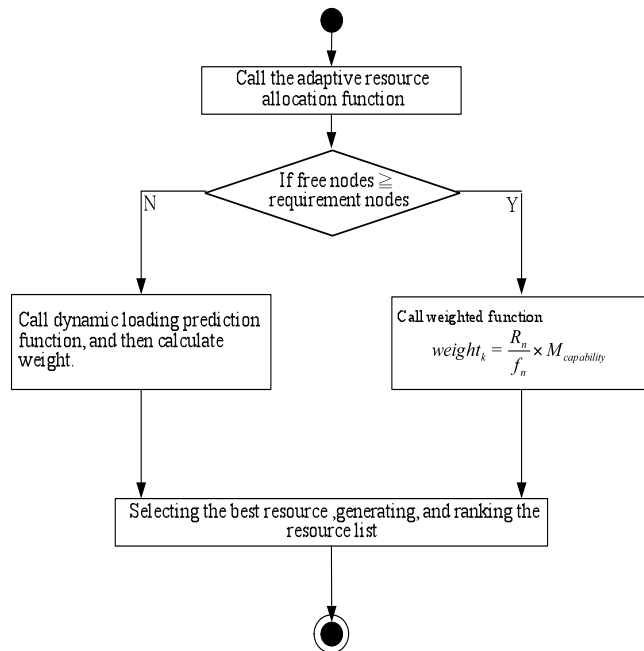


Fig. 5 The Logical Flow Chart of DLPS

Each step of DLPS algorithm is described as the following. Step 1: Process users' criteria and job

requirements from job script, job template, or Portal form specification, including the High-Performance Linpack Benchmark, data set, execution programs, and queue type, etc. Step 2: Make communicate with each MC Slave of clusters to get the static and dynamic resource information. Step 3: (1) Store the features and status of each cluster into the Host Information Queue through the Information Manager; (2) Filter out unsuitable resource with the adaptive resource allocation function. Step 4: Compare these free nodes with required nodes. If current free nodes are enough, DLPS calls weighted function to calculate $weight_a$ and $weight_b$ (defined at **Definition 4**).

Definition 4: When free nodes fulfill required nodes, the designed weighted function is $weight_k = \frac{R_n}{f_n} \times M_{capability}$.

Where R_n means required nodes, f_n means free nodes and the $M_{capability}$ means the capability of each computing resources.

Step 5: If current free nodes are not enough for fulfilling, DLPS calls dynamic loading prediction function with two methods to calculate $Weight_c$ (defined at **Definition 7**), including *EstBacklog* and minimum *Job Expansion Factor* methods.

Definition 5: The *EstBacklog* means estimated backlog of queued work in hours. The general *EstBacklog* form is shown as the following equation (2):

$$EBL_i^* = \left(\frac{QueuePS \times CPUAccuracy}{TotalJobsCompleted} \right) \times \left(\frac{TotalProcHours \times 3600 \times AvailableProcHours}{DedicatedProcHours} \right) \quad (2)$$

EBL_i^* , it means the i th *EstBacklog* time. *QueuePS* is the idle time of queued jobs, *CPUAccuracy* is the actual run time of job, the *TotalJobsCompleted* is the number of jobs completed, the *TotalProcHours* is the total number of proc-hours required to complete running jobs. The *AvailableProcHours* is the total proc-hours available to the scheduler, and the last variable, *DedicatedProcHours*, is the total proc-hours made available to jobs.

Some of above values are from the system historical statistic values of queuing system loading and the others are from real-time queuing situation. The output is divided into two categories, running and completed. The Running statistics include information about jobs that are currently running. The completed statistics are compiled using historical information from both running and completed jobs. Therefore, the EBL_i^* can forecast the backlog of each computing site with above information.

Definition 6: The *job expansion factor* subcomponent has an effect similar to the queue time factor but favors shorter jobs based on their requested wallclock run time. In its canonical form, the job expansion factor metric is calculated by the information from local queuing system which described as the equation (3):

$$JEF_i = \frac{QueuedTime + RunTime}{WallClockLimit} \quad (3)$$

Definition 7: After getting *EstBacklog* and job expansion factor, the *Weight_c* metric is calculated by the following equation (4) :

$$Weight_c = \lambda \times \frac{JEF_i}{\sum_{i=1}^n JEF_i} + (1 - \lambda) \times \frac{EBL_i^*}{\sum_{i=1}^n EBL_i^*} \quad (4)$$

Where λ is the system modulated parameter which can be obtained from numerous trials. The *EstBacklog* can be respected the dynamic situation of queuing system generally. Therefore, it always uses the higher λ value.

Consequently, the Step 6: Calculate the minimum time of total deliver or response time.

3.5 Power Saving Mechanism

We developed an approach to reduce energy utilization in local cluster. We do this work on the integration of resource management system and remote power management system that aims at reducing power consumption such that they suffice for meeting the minimizing quality of service required by local cluster. In particular, our approach relies on recalling services dynamically onto appropriate amount of the machines according to user's job request and temporarily shutting down the machines after finish in order to conserve energy. As shown in Figure 6, the power saving mechanism will wake up every minute to check job queue if there exist jobs, and make sure the machines 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 [18] protocol or IPMI [19]. After the job completes, the power saving mechanism powers the machines down. Our implementation currently relies on checking the local queuing system (i.e. Torque [14,15]) job pool and then decides to shut down which compute nodes when no new job was submitted. By powering off idle machines, it can significantly save more energy than always keeping all machines running.

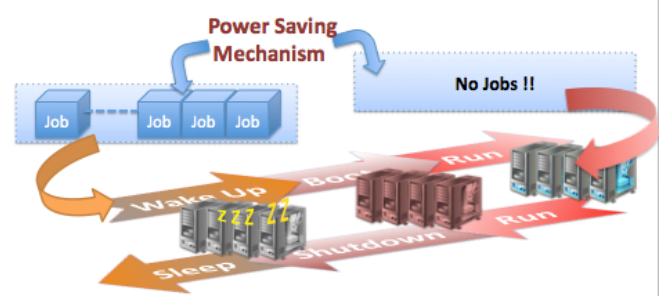


Fig.6 Scenario of Power Saving Mechanism

4 Performance Evaluation

4.1 Experimental Environment

In order to test the efficiency of the developed MCRM with DLPS scheduling algorithm, we execute a computational fluid dynamics (CFD) program with message passing interface (MPI) on a heterogeneous research testbed. We adopt the NCHC testbed, including Nacona, Opteron, and Formosa Extension two (Formosa Ext. 2) clusters. Those main environment characteristics are summarized in table 1. And we also measure the High-Performance Linpack Benchmark of these clusters. The R_{max} means the maximal performance in Gflop/s for each problem run on a machine, and the R_{peak} means the theoretical peak performance Gflop/s for the machine. Hence, users can choose the higher R_{max} value or set the criteria of computing power when users submit jobs, according to the information, in table 2.

Table 1 Summary of Environment Characteristics of NCHC Resources

Resource	CPU Model	Memory (GB)	CPU Speed (MHz)	#CPUs	Nodes	Job Manager
Nacona	Intel(R) Xeon(TM) CPU 3.20GHz	4	3200	16	8	Torque
Opteron	AMD Opteron(tm) Processor 248	4	2200	16	8	Moab
Formosa Ext. 2	Intel(R) Xeon(TM) CPU 3.06GHz	4	3060	22	11	Maui

Table 2 High-Performance Linpack Benchmark of NCHC Resources

High-Performance Linpack Benchmark	Nacona Cluster	Opteron Cluster	Formosa Ext.2 Cluster
R_{max} (Gflops)	46.791424	34.08	75.447
R_{peak} (Gflops)	102	70	134.64
Number of CPUs	16	16	22
The Efficiency of CPU (Gflops/CPU)	2.924	2.13	3.429

4.2 Experimental Scenario

The preliminaries of experiment are needed to set up, including the start time of jobs, the convergence of MPI matrix, and the number of required computing CPUs. The above preliminaries are generated by normal distribution. Therefore, we generate several execution MPI programs with 2, 4, and 8 CPUs randomly. The evaluation also has been performed on three clusters with three experiment models, including 4096*4096 matrix with 2 CPUs, 4096*4096 matrix with 4 CPUs, and finally the last model is the 8192*8192 matrix with 8 CPUs.

The performance of DLPS is compared job scheduling algorithm with several algorithms, such as Round-Robin, Short-Job-First (SJF), Big-Job-First (BJF), and First-Come-First-Serve (FCFS). We submitted testing jobs, which were generated randomly with the synthetic models as the Fig. 7 is shown. The vertical axis is the value of min makespan (seconds) and the horizontal axis is the number of jobs. The makespan of MCRM with DLPS job scheduling algorithm is notably less than other algorithms; especially when a huge number of jobs are submitted. Therefore, the objective function of DLPS approaches the minimized makespan. The dynamic loading prediction characteristic of presented MCRM is proved to be better in this experiment.

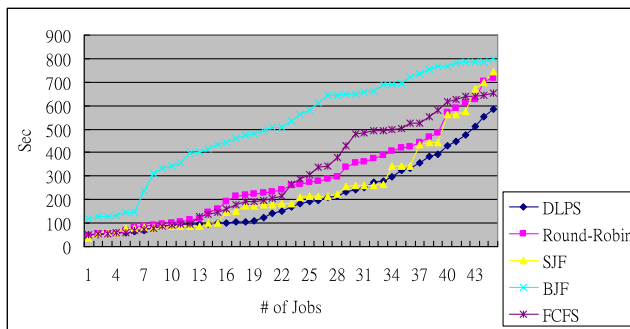


Fig. 7 Compare Makespan of DLPS with Other Algorithms

When a small number of jobs are submitted, the efficiency of DLPS may be worse than other algorithms, especially for SJF and FCFS. This situation is reasonable, because small jobs are easy consumed by SJF and FCFS. When the number of jobs is increasing, the developed DLPS is absolutely better than SJF and FCFS, because the notable drawback of SJF and FCFS happens, which the large numbers of jobs are queued inefficiently in the local scheduler of cluster. Comprehensively the above efficiency figures, the best efficiency of DLPS occurs at full usage of each cluster.

5 Conclusions Conclusion and Future Work

The Multi-Cluster Resource Manager (MCRM) takes a step further in the direction of establishing large computing sites for the multi-cluster computing system.

In this paper, we presented the MCRM, which can satisfy the users' requirements, including the hardware specification, specific software, and the High-Performance Linpack Benchmark results. And then, the MCRM can automatically select the most appropriate physical computing resource. With the features of automatic scheduling and dynamic loading prediction provided by the Scheduling Module, the users are no longer required to select the execution resource for the computing job. Furthermore, the power saving mechanism we developed. It wakes up the idle machines when computing power is needed and then powers them down when the job is done. It reduced power consumption successfully.

Instead, the MCRM will provide an automatic selection mechanism, which integrates both static information and dynamic information of resources, to meet the demand from the user's jobs. According to the pervious experiments, the dynamic loading prediction job scheduling has better efficiency and performance than other algorithms; especially the huge numbers of job are submitted into the computing cluster. Finally, we obtain an important property that the algorithm is appropriate to deal with large amount of jobs in multi-cluster system.

6 References

- [1] R. Al-Khannak, and B. Bitzer, "Load Balancing for Distributed and Integrated Power Systems using Grid Computing," International Conference on Clean Electrical Power (ICCEP), 22-26 May, 2007, pp. 123-127.
- [2] Hsu, C.-H.; Lo, T.-T. and Yu, K.-M. "Localized communications of data parallel programs Qn multi-cluster grid systems," European Grid Conference (EGC'05), Lecture Notes in Computer Science, vol. 3470, pp. 900-910, June 2005.
- [3] Javadi, B.; Akbari, M. and Abawajy, J. "Performance analysis of heterogeneous multicluster systems," in Proceedings of the International Conference on Parallel Processing (ICPP'05), pp. 493-500, 2005.
- [4] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund, "Dynamic Matching and Scheduling of a Class of Independent Tasks onto Heterogeneous Computing Systems," In the 8th IEEE Heterogeneous Computing Workshop (HCW'99), 1999, pp. 30-44.
- [5] J. Schopf, "A General Architecture for Scheduling on the Grid," *Journal of Parallel and Distributed Computing*, special issue, April 2002, p. 17.
- [6] A. Othman, P. Dew, K. Djemame and I. Gourlay, "Toward an Interactive Grid Adaptive Resource Broker," *Proceedings of the UK e-Science All Hands Meeting*, Nottingham, UK, September 2003, pp. 4.

- [7] M. Grajcar, "Strengths and Weakness of Genetic List Scheduling for Heterogeneous Systems," Application of Concurrency to System Design, 2001. *Proceedings. 2001 International Conference*, 25-29 June 2001, pp. 123-132.
- [8] Barry G. Lawson, and E. Smirni, "Multiple-queue Backfilling Scheduling with Priorities and Reservations for Parallel Systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 29, Issue 4, March 2002, pp. 40-47.
- [9] N. Fujimoto, and K. Hagihara, "A Comparison among Grid Scheduling Algorithms for Independent Coarse-Grained Tasks," *Symposium on Applications and the Internet-Workshops*, Tokyo, Japan, 26 – 30, January, 2004, p. 674.
- [10] N. Fujimoto, and K. Hagihara, "Near-optimal Dynamic Task Scheduling of Independent Coarse-grained Tasks onto a Computational Grid," *In 32nd Annual International Conference on Parallel Processing (ICPP-03)*, 2003, pp. 391–398
- [11] O. H. Ibarra, and C. E. Kim, "Heuristic Algorithms for Scheduling Independent Tasks on identical Processors," *Journal of the ACM*, vol. 24, no. 2, 1977, pp. 280–289.
- [12] Yi-Lun Pan, Chang-Hsing Wu, and Weicheng Huang, "A Grid Resource Broker with Dynamic Loading Prediction Scheduling Algorithm in Grid Computing Environment," in *Proceedings of The 2008 International Conference on Grid Computing and Applications(GCA'08)*, pp 28-34, 2008.
- [13] <http://www.wxwidgets.org>
- [14] <http://www.openssh.org>
- [15] <http://www.clusterresources.com/pages/products/torque-resource-manager.php>
- [16] <http://www.openpbs.org>
- [17] <http://www.clusterresources.com/pages/products/moab-cluster-suite.php>
- [18] Wake-on-LAN,
<http://en.wikipedia.org/wiki/Wake-on-LAN>
- [19] IPMI,
<http://www.intel.com/design/servers/ipmi/index.htm>

Developing an Intelligent Layer for Automatic Parallel Detection Implemented on Different High Performance Computing Platform

Mohamed Ahamed Mead¹, Hesham ElDeeb², Salwa Nassar²

¹Computer Science, Mathematics Department, Faculty of Science, Cairo University, Cairo, Egypt.

²Computer & Control Department, Electronic Research Institute, Cairo, Egypt

Abstract - In this paper, an automatic parallelization tool for C code, named Intelligent Automatic Parallel Detection Layer (IAPDL), is presented. It generates parallelized MPI code, and OpenMp code from the sequential code; at the loop level, to be executed on a cluster platform and multicore platform respectively. In addition to, a tool that uses a new approach to choosing loop transformations, called Intelligent Loop Transformation Selector (ILTS), is developed. It is designed as an integrated part in IAPDL. The selection process of appropriate loop transformation was accomplished intelligently; a Kohonen's Self-Organizing Map (SOM) neural network is used to select the appropriate loop transformation or sequence of them.

Keywords: Automatic parallelization, Cluster of Workstations, Multicore Processor, Dependence Analysis, Loop Transformation, Neural Networks.

1. Introduction:

Parallel computers provide high computations power, needed by many existing applications. They are widely used to overcome on the limitation of traditional computers that impede solving of large problems. Cluster systems have the same architecture as Distributed Memory Multicomputer systems, while multicore processor has an architecture like shared memory multiprocessors[1][2]. Cluster and multicore systems involve a tremendous programming effort on the programmers. Automatic parallelization process overcome on these difficulties, it converts the old designed code and the modern code to parallel form. Automatic parallelization of a program is generally achieved using parallel tools and compilers that vary in design, results, optimization techniques, and generality.

Parallelism has been exploited at multiple levels, Instruction level, Loop level, Procedure level and so on. Since most of parallelism in numeric and scientific programs is exist in loops. As well, loop level execution is time consuming and it is easy to implement, loop level is chosen to be our parallelism level in this paper.

Many researches in parallelizing tools and compilers have been introduced, but their functionality, performance, and scope is still very limited[3]. However, recent developments in parallelizing tools and compilers are attempted to provide full automatic parallelization[4].

This paper presents a design and an implementation for a parallelizing tool, named IAPDL for a cluster and multicore platforms. IAPDL get a sequential C code and produces as output a parallelized MPI code, and OpenMp code at the loop level, to be executed on a cluster and multicore platforms respectively. As well, based on neural networks techniques, IAPDL introduce the intelligent loop transformation selection techniques as an approach to select the appropriate loop transformations. Inserting Neural Networks offer intelligent transformations selection to reduce or eliminate the dependencies and maximize the parallelization in the sequential code. Although there are many systems that are closely related to our work, most of them fails to provide a full automatic parallelization of sequential code in most cases and concentrates only on special forms of code. In addition to, it select and apply loop transformation individually, rather than on their interplay when combined. IAPDL consists of 4 phases; Front end, analysis, Intelligent Transformation, parallel code generation.

IAPDL should be able to facilitate the task of the programmers by the following contributions:

- 1- Give intelligent decision about the appropriate loop transformations that suitable for the code.
- 2- Give an equivalent parallel code to sequential code at the loop level for cluster and multicore platforms.

For simplifying the problem, the accepted code by IAPDL has some limitations. All accepted loops must be either single or perfectly double nested. However, the IAPDL can accept a non-perfectly multi-level nested when dependence is simply disproved. IAPDL front end will be introduced in the next section. The Dependence Analyzer and the Intelligent Loop Transformation Selector are introduced in section 3 and 4 respectively. The IAPDL suggestion parallel code is showed in section 5. The performance evaluation of IAPDL will be presented in section 6. Finally, the discussion and results will be concluded in section 7.

2. Front End: Lexer And Parser

The front end of IAPDL is implemented by using a parser generator. A parser generator is a tool that reads a grammar specification and converts it to a program that can recognize matches to the grammar[5]. Many parser generating tools exist; it can be used for parsing c code and

any other language. Since the IAPDL will be implemented using the Java programming language then a parser generator must producing output in java language. A variety of parser generator that generate parser in java language are considered. CUP [6], SableCC [7], ANTLR [8], Coco/R [9] or JavaCC[10] are the most famous Java based parser generator tools.

SableCC, ANTLR, and JavaCC are the most suitable for our work. Javacc is easy to learn and easy to use, as well it is very stable, then JavaCC and the preprocessor JJTree tool is used in this thesis to generate the lexer, parser, and parse tree of C language.

The IAPDL required full information about the sequential code to generate the corresponding parallel code. During the front end phase the IAPDL gathers information from sequential code that will stored in a database in a form like a control-flow graph. This information include, variables, constants, statements, procedures, functions, and scope. The database allows the future extension of the IAPDL without major modification. The analysis and subsequent phases work directly on this database system.

3. Dependence Analyzer

Dependence analysis determines the dependence relations between different parts of the program[11]. Automatic detection of parallelism in sequential program relies mainly on data dependence analysis process[12][13][14][15][16][17][18]. The most important types of data dependence relations are: flow, anti, output dependence.

IAPDL performs the data dependence analysis in 3 Steps. Data dependence for scalars is determined in the first step, while the other two steps are determine the data dependence analysis for arrays. IAPDL manages the choosing and applying dependence testing based on classifying pairs of subscript references as in [19]. This classification allows the IAPDL to choose the most efficient test for a given pair of references. Since most subscripts are ZIV and SIV, the dependence analyzer accepts only the ZIV and SIV subscripts as a start point for unlimited dependence analysis.

4. Intelligent Loop Transformation Selector (ILTS):

Existence The loop carried dependence prevents the parallelization of the loop, which leads us to loop transformation techniques. Loop transformation techniques is an important compiler optimization that enhance the parallelism in program and data locality [20] [21] [22][23][24][25]. The loop transformations change only the execution ordering while the overall computation remains essentially the same [24].

There are many types of loop transformations. Some of these transformations aid to enable other loop transformations and do not result in any optimization by themselves. The most important loop transformation types

are, interchange, skewing, reversal, distribution, scalar expansion, fusion, tiling, unrolling and peeling[24].

Inserting neural networks technique will facilitate and improve the selection of Loop Transformation process. In this paper, ILTS uses kohonen's SOM network as a neural network model to cluster sequential loops according to appropriate loop transformation that may be applied on these loops. The input/output of ILTS is showed in figure (1).



Figure 1. Architecture of IAPDL

Now, the loop transformations that will be manipulated by the (ILTS) will be focused. There are many loop transformations, but with respect to dependencies in real code, the transformation that necessary to optimize code are few[26]. The transformations that will be chosen in (ILTS) are dependent on effectiveness, and its interplay with the other transformation that will be chosen. In this paper, the loop transformations are used individually or in groups as follows:

Individually

- 1- Distribution
- 2- Interchange
- 3- Scalar Expansion

Groups

- 4- Scalar Expansion followed by Distribution
- 5- Statement Reordering followed by Distribution
- 6- Skewing followed by Interchange

The first step in our work is to extract features vectors from each sequential loop to be passed to the network. Features vector that extracted from a loop must be high information content and affecting in selecting loop transformations. Since most loop transformation try to eliminate the loop carried dependence relations, then it is considered the main parameter that influences in selecting loop transformations. Each loop carried dependence relation and some loop attributes are transformed by the dependence analyzer of IAPDL into a features vector that can be used as input to the ILTS.

Type of data dependence relations, flow, anti, and output are influence in choosing loop transformations. Also, direction vector is considering the important dependence information that influence in loop transformation selection[27]. Cyclic loop-carried dependence and lexically order of the dependence relation are important information in applying loop distribution and statement reorder transformations. Dependence level is considered as the minimal abstraction for the loop reversal transformation[22].self dependence and the number of statements enclosed in a loop affecting in cancellation some loop transformations. The main features that mentioned above dependent on the type of loop transformations manipulated in ILTS. These features can be extracted from every loop carried dependence relation and mapped into

ILST. These features are affecting in determining the appropriate loop transformations as illustrated in performance evaluation process.

The above features forming a 15 components vector that represents a loop and its carried dependence relations. The details of ILST is illustrated in [28]

5. The suggestion parallel code:

Finally the code generator phase in IAPDL generates the MPI and OpenMP code that will be executed on cluster of workstations platform and multicore platform respectively. This phase interacts with the database that is created through the previous phases to recreate the sequential code in parallel form. It inserts the library routines for MPI and compiler directives, library routines, and environment variables for OpenMP.

5.1. OpenMP Code:

OpenMP was designed for shared memory machines and is considered an easy method for threading application [29][30]. The IAPDL approach to generate a parallel code for multicore can be summarized as follows:

1- Threads creation and work scheduling over them

Generating efficient OpenMP codes for multicore in this paper depend on loop-level parallelism; since the full potential of OpenMP will be acquired from threading the most time-consuming loops. Although the OpenMP 3.0 can handle the task parallelism, this feature will be implemented in the further version of the IAPDL.

The loops, except some cases, that has loop carried flow dependence can not be parallelized by the IAPDL; only the loop with independent iterations can be parallelized.

The default number of threads is currently set to the number of available cores, and this what we will use in IAPDL. The directive "#pragma omp parallel for" distributes the loop iterations to all threads. There are four types of scheduling the loop iterations construct, static, dynamic, guided, and runtime. The scheduling process in IAPDL are achieved statically, all the threads are allocated iterations before they execute the loop iterations. Static scheduling is the best choice in our work for many reasons. In dynamic, guided, and runtime scheduling, the performance can be adversely affected by changing the chunk size [29]. In the contrary, in static scheduling changing the chunk size has a little effect on the performance. In addition, in most cases there is no variation in compute time among loop iterations, which lead to static schedule.

2- Managing shared and private data

For program correctness, all data must be managed to Understanding which data is shared and which is private. Shared variables are shared among all threads, while private variables vary independently within threads. The IAPDL tells the compiler which pieces of memory should be shared among the threads and which pieces should be kept private. It is able to do this by inserting a set of clauses.

In the generating parallel code the loop index must be private, since the loop iterations are distributed over the threads in the team. IAPDL mark the variable to be private in two cases:

- Loop index
- Variables that updated within a parallel region

Without passing the updated variables from the last loop iteration to the master threads, the behavior of the program will be changed. The IAPDL inserts the lastprivate clause to ensure that the master thread has the correct values of these variables. In figure (2) the variable x must be declared as lastprivate as illustrated in figure (3). In addition to, the loop index may be used after the loop which leads us into lastprivate clause rather than private clause.

```
for (i=0;i< num_steps; i++){
x = (i+0.5)*step;
}
```

Figure 2. Simple loop

```
#pragma omp parallel for lastprivate(i,x)
for (i=0;i< num_steps; i++){
x = (i+0.5)*step;
}
```

Figure 3. OpenMP code for simple loop

3- Reduction Calculation

Consider the statement in figure(4); handling this type of statement is very important because it is occurred in most scientific code.

```
for (i = 0; i < n; i++) {
sum = sum + a[i];
}
```

Figure 4. Reduction Operation

OpenMP has a specific clause; reduction clause, to handle some forms of recurrence calculations. Only the operations and the variables that will hold the result values will be identifying. The variables must be scalar, and shared in the enclosing context. When the IAPDL discover these cases, it inserts the reduction clause as illustrated in figure(5). Each thread has a private copy of the variable Sum, and at the end each of them has the final summation of the variable.

```
#pragma omp parallel for reduction(+:sum)
for (i = 0; i < n; i++) {
sum = sum + a[i];
}
```

Figure 5. OpenMP code for Reduction Operation

5.2 MPI Code:

MPI is a language-independent communications protocol, designed for distributed-memory platform [31][32]. Each processor has access to its own memory, and communicates information by sending and receiving data between them. The IAPDL approach to generate a parallel code for cluster platform can be summarized as follows:

1- Processes creation and work scheduling over them

Generating MPI code in IAPDL involves issues such as computation distribution, data consistency among processors. IAPDL partitions the loop iterations into parallel tasks according to data dependence analysis. It uses the data dependence analysis information to decide if any loop can be parallelized or not. The number of iterations of parallel loop is computed dynamically by subtraction the start value from end value, which is divided by the number of processors to get the available number of tasks. As well as the IAPDL assign each sequential code before and after any parallelized loop to all processors to minimize the data communication and synchronization overhead.

2- Data Consistency between Processes

Using the work scheduling schemes of IAPDL, after any parallelized loop the data that updated in the local memory of one process must be needed by another. In this case, processes need to communicate via message passing. The fundamental issue that needs to be addressed is the determination of the data that need to be transferred. In our approach every updated scalar variable is Broadcasted from the process with rank $nproc-1$; the last process, to all other processes. MPI_Bcast routine is used in this case to ensure that all processes have consistent values of scalar variables. In addition to, by using the MPI_Allgather routine all processes send their local arrays to all processes.

3- Reduction Calculation

Handling some forms of recurrence calculations is very important; in MPI the routines MPI_reduce and MPI_Allreduce perform that function. MPI_reduce accumulates the final results in the master process, while MPI_Allreduce accumulates the final results in all processes. Since all processes execute the same sequential part of the generating code, then the reduction results must be available to all of them. The routine MPI_Allreduce performs that function in IAPDL.

6. Performance Evaluation of IAPDL

In order to expose the effectiveness of our work the IAPDL has been evaluated on several case studies. In this section five study cases are considered in the evaluation process to measure the correctness, and the performance of IAPDL. The first case study, calculating pi, is a simple application shows how the IAPDL manipulate the reduction calculation. The second is the matrix multiplication included since it is used in a variety of famous applications. While the third case study is a simple loop extracted from [33] that exposing the benefit of ILTS in detecting the appropriate loop transformations intelligently. Finally, the other two case

studies require heavy and complex computations (numerical techniques), as well as, it has an extensive parallelization opportunity.

Two different parallel architectures are used for the results in this paper.

1- Intel Core 2 Quad Q8300 quad-core CPU clocked at 2.5 GHz (1333 MHz front side bus) with a 4MB of L2 cache.

2- A Linux and homogeneous cluster is built in Electronic Research Institute (ERI), it is formed of ten machines (one master node and 9 slave nodes).

Intel's C compiler, ICC 11 with OpenMP 3.0 support, is used to compile the OpenMP codes on the Intel core, while GCC compiler is used to compile the MPI codes on cluster platform.

6.1 Case Study 1: PI Calculation

```
for (i=0;i< num_steps; i++){
    x = (i-0.5)*step;
    sum += 4.0/(1.0+x*x);
}
```

Figure 6. Calculating PI (Serial)

As illustrated in figure (6), although there are a scalar loop carried dependence relations by both variable x and sum , the IAPDL overcome on these dependence relations by privatization and reduction recognition analysis. The IAPDL is used to fully automatically parallelize this loop for Multicore and cluster platform. The loop iterations have to be distributed to several threads for OpenMP code and several processes for MPI code.

The recursive use of the variable sum , which is read and modified with each loop iteration, can be implemented by using the reduction clause for OpenMP code and the routine MPI_Allreduce for MPI code. In addition to, since the variable x is modified and read then the value of variable x at last iteration must be copied to the original variable object in OpenMP, and broadcast to all processes in MPI.

The benefits of the proposed IAPDL for the first case study are shown in figure (7). The speedup of the generated OpenMP and MPI code running on multicore and cluster systems respectively, are plotted with respect to 10^8 and 10^9 number of iterations. It indicates that the IAPDL code improves the execution time with respect to sequential code on both cases.

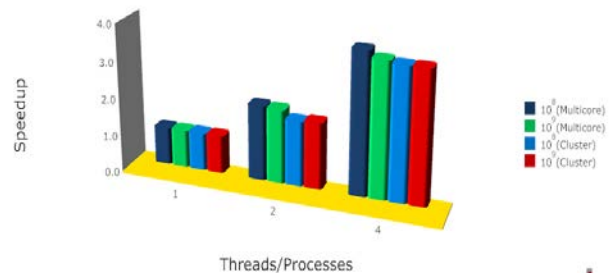


Figure 7. Speedup Results for PI calculation with 10^8 and 10^9 iterations

6.2 Case Study 2: Matrix Multiplication

Matrix multiplication is used in many scientific applications like image and signal processing. Since most of these applications have matrices with large sizes, the parallel implementation of matrix multiplication is an efficient way to parallelizing these applications.

The IAPDL discovers that the outer loop has no any loop carried dependence, and then this loop can be parallelized. Simply the OpenMP code can be generated by inserting the compiler directive "#pragma omp parallel for" before the outer loop and declare the variables i, j and k as lastprivate. On the other hand MPI code is harder to generated, the loop iterations have to be distributed to several process. In addition to, the data must keep consistency at all processor. According to IAPDL approach for MPI code, all the parallel processes must have the updated data after the parallel part. The routine MPI_Allgather is used to handle this case.

The benefits of the proposed IAPDL for the matrix multiplication problem are shown in figure (8). The speedup of the generated OpenMP and MPI code running on multicore and cluster platforms respectively, are plotted with respect to matrix sizes 1000X1000, 2000X2000, and 3000X3000. It indicates that the IAPDL code improves the execution time with respect to sequential code on both cases.

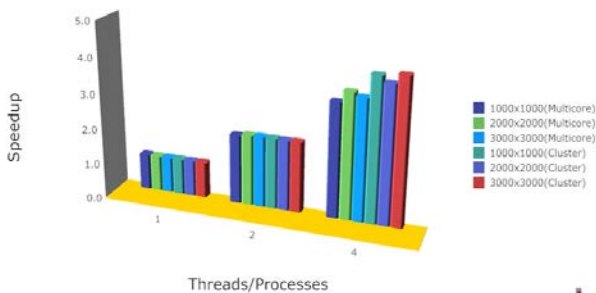


Figure 8. Speedup Results for matrix multiplication with different sizes

6.3 Case Study 3: Simple Loop Benchmark

In order to expose the effectiveness of ILTS tool, one loop from [33] are considered in the evaluation process to measure the correctness, and the performance of ILTS. Although this loop is simple, it reveals the benefit and correctness of ILTS tool. Figure (9) illustrates the sequential code of this loop.

```

for(i = 1; i < n-1; i++)
{
    a[i] = b[i-1] + c[i] * d[i];
    b[i] = b[i+1] - e[i] * d[i];
}

```

Figure 9. Simple Loop Benchmark

This loop is analyzed by IAPDL phases to extract the features that will be passed to the ILTS. Two features vectors are obtained from this analysis, one vector for each loop carried dependence. Each vector is mapped to the ILTS

to select a neuron on the map that represent the appropriate loop transformations.

The first vector is mapped to the neuron (4,8), while the second mapped to the neuron (4,5). As illustrated in[28] the appropriate loop transformation for this loop is distribution transformation, or Reordering transformation followed by distribution transformation or scalar expansion followed by distribution transformation. The reordering transformation followed by distribution transformation is the best choice. The result of these transformations is legal for this case as illustrated in figure (10), which reveals the correctness of ILTS.

```

for(i = 1; i < n-1; i++)
{
    b[i] = b[i+1] - e[i] * d[i];
}
for(i = 1; i < n-1; i++)
{
    a[i] = b[i-1] + c[i] * d[i];
}

```

Figure 10. Reorder statement followed by Loop Distribution Transformation for figure 9

6.4 Case Study 4: Solves the 2D Steady State Heat Equation

The sequential code of this problem is analyzed and parallelized by IAPDL, but the integrated tool; ILTS, detects that there is a chance to apply the loop distribution transformation. The loop distribution will be applied by user then the optimized sequential code is passes again to the IAPDL. Applying this loop transformation will be increase the opportunity of parallelization. Figure (11) illustrates a comparison between the speedup obtained using the generated OpenMP code before and after applying loop distribution transformation for different size 2D grid. This comparison reveals the benefit and correctness of ILTS tool.

With respect to MPI code, there are many reduction operations in this code that was implemented by using MPI_Allreduce routine. Although, parallelization this part of code will decrease the execution time, but the overall execution time is increased. According to IAPDL approach for cluster platform, after each parallelized loop all updated arrays at every process are sent to all other processes by using the MPI_Allgather routine. However, the MPI_Allgather routine requires a higher bandwidth that may be limit the performance of parallel programs, and in some cases the parallel execution time may be become greater than sequential execution time. In this problem the parallel execution time become greater than sequential execution time due to a bad balance between computation and communication.

A user interaction is required to remove the data exchange between processes from the generated MPI code that does not required and passing only the boundary rows between processes.

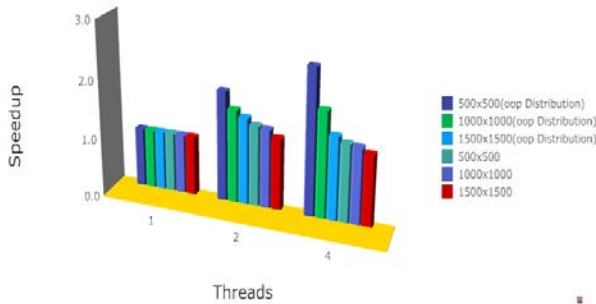


Figure 11. Speedup Results for 2D Heat Equation before and after Applying Loop Distribution Transformation on Multicore

6.5 Case Study 5: Specific Absorption Rates Calculation

In this case study, 3D Finite Difference Time Domain (FDTD) is used to calculate the Specific Absorption Rate (SAR) distribution on the human head due to radiation from handheld wireless devices[34]. This problem involves a very heavy computation that needs high computing power. Parallel implementation for this problem solves the bottleneck in running this type of application.

Many loops in this application do not have any loop carried dependence, and then these loops can be parallelized. IAPDL detect most of loops that can be parallelized. The generated OpenMP code has been a reasonable speedup, while the generated MPI code has been a negative speedup. As stated in the previous case study, the imbalance between communication and computation in the generated MPI code that imposes a limit on cluster platform performance.

Again, a user interaction is required to remove the unrequited data exchange between processes from the generated MPI code. Manually this problem is solved for cluster platform in[35], its speedup is compared with the speedup of the generated OpenMP code on multicore platform as shown in figure (12).

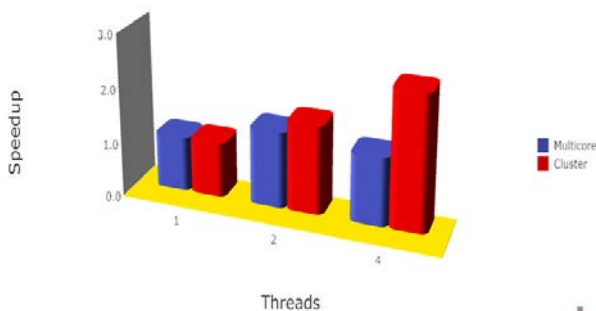


Figure 12. Speedup Results for Specific Absorption Rates Calculation

From this result we can conclude that, the magnitude of the performance enhancement caused by 4 threads is less than that of the 2 threads. In addition to, the cluster platform gives a better speedup when compared with multicore platform; especially with growing the number of processes.

7. Conclusion and future work

The goal of this work is to produce an efficient tool; Intelligent Automatic Parallel Detection Layer (IAPDL) to facilitate the programming on a cluster platform and multicore platform, and introduce a significant step toward achieving automatic and intelligent loop transformation detection. In this paper, only the loops without loop carried flow dependence were parallelized. A kohonen neural network is used to categorize the sequential loops with respect to the suitable loop transformations to save the time of expert.

A famous scientific\engineering programs written in c code is selected to measure the proposed work. It is vary form simple to difficult and cover most coding structure. The performance measurements, given in this paper, showed that the execution time is significantly affected by using the proposed IAPDL relative to sequential one.

In the first and second study cases, the speedup is linear or close to linear. In addition to, the results explain that the IAPDL has the capability of selecting the appropriate loop transformations besides selecting the optimum sequence of them as in the third and fourth study case.

From the fifth study case results we can conclude that

- 1- The core 2 quad with 2 threads enabled optimal performance for SAR calculation programs. Since the cores in core 2 quad share the interface to front side bus as well as the L2 cache the memory-intensive job may be drop the performance by threads growing.
- 2- The main time consuming in our approach to generate MPI code is the MPI_Allgather operation which is used to produce global array in each process by gathering local array elements from other processes. When the computation time is large enough compared to MPI_Allgather operation time as in study case 2, we get a very good speedup. On the other hand, the imbalance between communication and computation as in fourth and fifth study case limit the performance of cluster platform. By user interaction to modify the generated MPI code we get a suitable speedup, compared to the speedup of OpenMp code running on multicore platform.

The new interconnection technologies such as Myrinet, Quadrics and Infiniband can solve the MPI_Allgather communication bottleneck[36][37]. In the future, IAPDL will give a full automatic MPI code with reasonable speedup by using these technologies in the cluster platform.

Finally we can conclude that, our approach is successful for most applications on multicore platform, but it is suffer from inadequate performance for cluster platform in some cases.

Currently, the IAPDL relies on user interaction to apply the appropriate loop transformations and then force the IAPDL to generate the parallel code for the optimized sequential code. In the future we try to automate this process. As well as, we can easily add the task parallelism in the next versions.

8. References

- [1] G. Blake, R. G. Dreslinski, T. Mudge, "A survey of multicore processors", *Signal Processing Magazine, IEEE*, Vol. 26, No. 6, 23 October 2009, pp. 26-37.
- [2] R. Buyya (editor): *High Performance Cluster Computing: Architectures and Systems*, Volume 1, ISBN 0-13-013784-7, Prentice Hall, NJ, USA, 1999.
- [3] D. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin. *Polaris: A New-Generation Parallelizing Compiler for MPP's*. Technical Report 1306, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., June 1993.
- [4] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores", *IEEE Computer*, vol. 42, no. 12, pp 36-42, Dec. 2009.
- [5] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, "Compilers: Principles, Techniques and Tools", Second Edition, Addison Wesley, 2006, ISBN-10: 0321486811.
- [6] S. E. Hudson. "The CUP parser generator for Java". (<http://www.cs.princeton.edu/~appel/modern/java/CUP>), 2008.
- [7] The Sable Research Group. *SableCC*. (<http://www.sablecc.org/>). 2008.
- [8] T. Parr, "Antlr, another tool for language recognition.", (<http://www.antlr.org/>). 2008.
- [9] The Compiler Generator *Coco/R*, User Manual, <http://ssw.jku.at/Coco/>, 2008.
- [10] Javacc Home, <https://javacc.dev.java.net>, 2008.
- [11] U. Banerjee., "Dependence analysis", Kluwer Academic Publishers, 1997.
- [12] S. S. Muchnick "Advanced Compiler Design and Implementation" Morgan Kaufmann; 1st edition 1997, ISBN-10: 1558603204.
- [13] R. Allen, and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form", *ACM Transaction on Programming Language and Systems*, Vol.9, No.4, p491-542, October 1987.
- [14] J. Sogno, "Analysis of Multidimensional Loops with Non-Uniform Dependences", *Advances in Parallel and Distributed Computing Conference*, p362-369, 1997.
- [15] W. Pugh, and D. Wonnacott "Constraint-Based Array Dependence Analysis", *ACM Transactions on Programming Languages and Systems*, Vol.20, No.3, p635-678, May 1998.
- [16] K. Psarris, and K. Kyriakopoulos, "An Experimental Evaluation of Data Dependence Analysis Techniques", *IEEE Transactions on Parallel and Distributed Systems*, Vol.15, No.3, March 2004.
- [17] Y. Yang, C. Ancourt, and F. Irigoien, "Minimal Data Dependence Abstractions for Loop Transformations", *International Journal of Parallel Programming*, Vol.23, No.4, p359-388, August 1995.
- [18] P. Boulet, A. Darte, G. Silber, and F. Vivien "Loop Parallelization Algorithm: From Parallelism extraction to code generation", *Journal of Parallel Computing*, Vol.24, No.3, p421-444, 1998.
- [19] G. Goff, K. Kennedy, and C. W. Tseng "Practical dependence testing". *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, pp. 15-29 1991.
- [20] P. Boulet, A. Darte, G. Silber, and F. Vivien, "Loop Parallelization Algorithm: From Parallelism extraction to code generation", *Journal of Parallel Computing*, Vol.24, No.3, p421-444, 1998.
- [21] J. Torres, E. Ayguade, J. Labarta, and M. Valero, "Loop Parallelization: Revisiting Framework of Unimodular Transformations", *4th Euromicro Workshop on Parallel and Distributed Processing*, p420-427, 1996.
- [22] M. Wolf, and M. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", *IEEE Transactions on Parallel and Distributed Systems*, Vol.2, No.4, p452-471, October 1991.
- [23] B. Chandramouli, J. Carter, W. Hsieh, and S. Mckee, "Cost-Model Driven Integration of Restructuring Optimizations", *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.
- [24] K. Shashidhar, M. Bruynooghe, F. Cathoor, and G. Janssens, "Geometric Model Checking: An Automatic Verification Technique for Loop and Data Reuse Transformations", *Electronic Notes in Theoretical Computer Science* Vol.65, No.2, p71-86, 2002.
- [25] A. Lim, and M. Lam, "Maximizing Parallelism and Minimizing Synchronization with Affine Transforms", *24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p201-214, 1997.
- [26] S. Amarasinghe, J. Anderson, M. Lam, and A. Lim "An Overview of a Compiler for Scalable Parallel Machines", *6th International Workshop on Languages and Compilers for Parallel*, p253-272, 1994.
- [27] D. Bacon, S. Graham, and O. Sharp, "Compiler Transformations for High-Performance Computing", *ACM Computing Surveys*, Vol.26, No.4, p345-420, Dec 1994.
- [28] M. Mead, H. Eldeeb, S. Nassar, "Automatic loop transformation selection with the aid of kohonen's self-organizing maps for parallelizing compilers", *PDPTA*, p850-856 (2008).
- [29] R. Chandra, L. Dagum, D. Kohr, D. Maydan, and J. McDonald, R. Menon "Parallel Programming in OpenMP", Morgan Kaufmann, 2000, ISBN-10: 1558606718.
- [30] C. Hughes, and T. Hughes, "Professional Multicore Programming: Design and Implementation for C++ Developers " Wiley Publishing, Inc., Indianapolis, Indiana, 2008, ISBN-10: 0470289627.
- [31] S. Vetter, Y. Aoyama, J. Nakano, "RS/600 SP: Practical MPI Programming", *IBM Redbooks*, 1999, ISBN-10: 0738413658.
- [32] G. Em Karniadakis, and R. M. Kirby II, "Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation", Cambridge Univ. Press, 2003, ISBN-10: 0521520800.
- [33] <http://www.netlib.org/benchmark/>, 2008.
- [34] E. A. Hashish, F.M.EL-Hefnawi, and A.Z.Elsherbeni. "A FDTD Scattered Field Formulation for Dispersive Media". *APS-2000*, Salt Lake. P. 248-251.2000.
- [35] H. Elsadek, H. Eldeeb, H. Abdallah, M. Desouky and N. Bagherzadeh. "Specific Absorption Rate Calculation using Parallel 3D Finite Difference Time Domain Technique". *WPRLDCOMP'08*. Las Vegas. 2008.
- [36] Y. Qian, "Design and Evaluation of Efficient Collective Communications on Modern Interconnects and Multi-core Clusters", Ph.D. Queen's University Kingston, Ontario, Canada, 2010.
- [37] G. Santhanaraman, T. Gangadharappa, S. Narravula, A. Mamidala and D. K. Panda, "Design Alternatives for Implementing Fence Synchronization in MPI-2 One-sided Communication on InfiniBand Clusters", *IEEE Cluster*, p 1-9 September 2009.

Go2ADLB: An Interface for Using ADLB Within Go

Ralph Butler, Chrisila Pettey, and Brian Manifold

Department of Computer Science
Middle Tennessee State University
Murfreesboro, Tennessee, USA

Abstract

Over the past five years extensive work has been done in developing an asynchronous, dynamic, load balancing library known as ADLB. This library makes it easier to rapidly scale parallel applications that require massive parallelism. The most notable use of ADLB to date has been the successful nuclear theory computations for carbon-12. These computations were done by the ASCR SciDAC Universal Nuclear Energy Density Functional project using over 131,000 processes. ADLB uses MPI to manage a workload while providing a simple interface for the application process. Until recently only C and Fortran interfaces have existed for the ADLB library. In this paper we present a Go interface (Go2ADLB) for the library.

Keywords: Asynchronous Dynamic Load Balancing, Go, High Performance Computing

1. Introduction

There are problems whose solutions require massive parallelism in order to be solved in a reasonable amount of time. Unfortunately it is frequently difficult to get these applications to scale to the hundreds of thousands of processes that are needed. One such application is the full calculation of carbon-12. Until 2006 researchers with the ASCR SciDAC Universal Nuclear Energy Density Functional project [3] had not run this problem with more than about 2000 processes. In order to do nuclear theory computations for carbon-12, they needed to scale the application to work with many more processes. By developing the Asynchronous Dynamic Load Balancing (ADLB) library, it was possible to scale the problem to over 131,000 processes in 2010 and do the full carbon-12 calculations [8].

The ADLB library [1, 8] was created to make it simpler to develop parallel code that would easily scale. Until now, ADLB has only been available for C and Fortran programs. In this paper we present a non-optimized version of ADLB written in Go (ADLB-Go), and a Go-to-ADLB interface (Go2ADLB) that allows programmers to write in Go and use the optimized production version of the ADLB library. In the next sections we briefly describe ADLB, the Go language, ADLB-Go, and the Go2ADLB interface for ADLB.

2. ADLB

While the idea of ADLB originated with the problem of how to scale a physics application, it was never intended to be used solely for that purpose. Instead, ADLB was developed as a model of how to dynamically balance the workload of any parallel program in such a way as to allow it to easily scale to whatever number of processes was needed. Since its introduction, there have been reports of various different applications that have used ADLB even though the library has not been widely publicized. The resounding success of ADLB with the physics application has earned it a place in the 2012 congressional budget of the Department of Energy's Office of Advanced Computing Research [5].

The ADLB programming model is deceptively simple but completely scalable, and as noted previously it has been proven to over 131,000 processes. In a nutshell, ADLB achieves scalability by combining the two concepts of dynamic load balancing and work stealing. More specifically, ADLB uses MPI to create a group of server processes whose sole responsibility is to manage a distributed work queue. From the application process point of view, the work queue is a single shared pool. Application

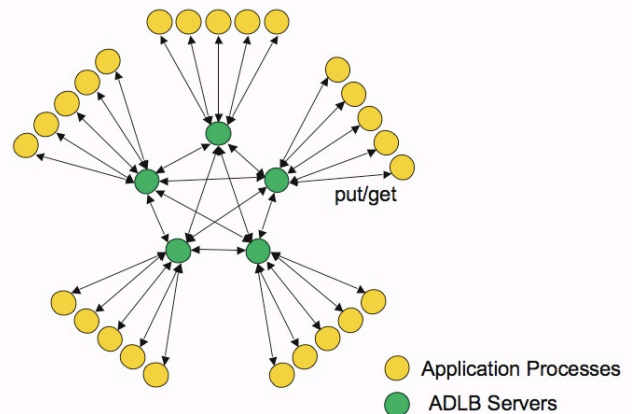


Figure 1. Interactions between ADLB server processes and application processes [1]

processes use the pool as a shared resource where they can put work and reserve/get work. Behind the scenes, the ADLB servers are moving data around to balance the

workload. Figure 1 (taken from [1]) illustrates the interaction between the ADLB server processes and the application processes. In one large run of the previously mentioned physics application, there were 1.75 billion put/get operations transferring 338 terabytes of data.

While ADLB has gained some notoriety, it only has C and FORTRAN interfaces. These two interfaces were critical, since the primary general purpose languages for high performance computing are C and FORTRAN. However, there is a relatively new programming language, Go, that shows potential for becoming a very powerful parallel programming language. In the Tiobe Programming Community List which is updated monthly, Go ranked 19th in February 2011 [9]. Since Go is gaining in popularity, we decided to investigate developing a Go interface for ADLB. In the next section, we give a very brief description of Go.

3. Go

The Go programming language [4, 6] was released by Google in November 2009 after more than two years of design. According to the Go programming language website [6], Go is described as:

... an open source project to make programmers more productive. Go is expressive, concise, clean, and efficient. Its concurrency mechanisms make it easy to write programs that get the most out of multicore and networked machines, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of garbage collection and the power of run-time reflection. It's a fast, statically typed, compiled language that feels like a dynamically typed, interpreted language.

In other words, Go is a developer's dream. Not only is it easy to write code in Go, it is also quick to compile, and it has the execution speed of C. Ever since the Berkeley white paper, "The Landscape of Parallel Computing Research: A View from Berkeley" [2], it has widely been agreed that the future of computing must address the multi-core issue. In addition to its other stellar qualities, Go makes "it easy to write programs that get the most out of multicore" [6].

Since Go has the potential to become widely used among parallel programmers, we decided it was necessary to provide a Go2ADLB interface for the ADLB library in addition to the already existing Fortran and C interfaces.

4. ADLB-Go

Since we had no real experience with Go, we decided to learn by implementing a prototype of ADLB

totally in Go. We even implemented our own networking layer using Go's access to sockets rather than using MPI. We did not make the prototype fully functional, nor did we attempt to optimize it. Furthermore, we did not feel compelled to always use fully idiomatic Go. For example, the := operator permits one to essentially declare variables on the fly, and we chose instead to declare almost all variables much like in a C program.

Figure 2 contains a sample application program for ADLB-Go. In this simple example, each rank marshals a complex message consisting of multiple data types and puts the message to the work queue. Rank 0 then retrieves the messages and unmarshals them. A few things that one might notice about the code are that we did make the ADLB functions a part of a separate library, and we did marshal and unmarshal user level messages by hand via the Go-provided json library. We also demonstrate that we passed a message containing multiple data types.

However, to reiterate, this prototype was merely a learning experience, and is not intended to be a serious implementation.

5. Go2ADLB: The Go Interface for ADLB

Once we felt comfortable with the Go language, we began work on the Go wrappers for ADLB. This required us to write wrappers for both ADLB and MPI since ADLB relies on MPI for the networking. This proved to be a daunting task due to the fact that documentation on writing Go wrappers is still somewhat sparse, so one is largely dependent on the few examples provided to figure out how it works. Another problem that had to be overcome was that ADLB currently only supports static linking while most versions of MPI support dynamic linking. One point to be made is that it was not necessary to write wrappers for all of the MPI functions since ADLB only uses a tiny subset of the MPI functions.

A sample application program is listed in Figure 3. In this program, as in the standard ADLB, a number of ranks specified by the user of the program become servers. The rest of the ranks each put a large message to the work queue and retrieve a large message from the work queue. This may or may not be the same message. This example uses the same set of ADLB functions that you would expect to find in any ADLB program in any language. For example, there are *Init*, *Finalize*, and *Server* functions each of which is called only once. In addition there are the functions used to put data into and retrieve data from the shared work queue. These are *Put*, *Reserve*, and *Get_reserved*. Each of these is called as many times as necessary specific to the application.

```

package main
import (
    "fmt"
    "os"
    "json"
    "adlb"
)
type msgTyp1 struct {
    R int
    I int          // field names
    S string       // must begin with
    F float64      // a capital letter
}
func main() {
    var i, rc, myRank, numRanks int
    var msgtosend, msgrecvd msgTyp1
    var marshMsg []byte
    var targetRank, workType, workPrio int
    var userTypes, reqTypes []int
    var handle adlb.AdlbHandle
    fmt.Printf("INSIDE TADLB1 home %s rank %s numRanks %s\n",
        os.Getenv("FOO"), os.Getenv("ADLB_RANK"), os.Getenv("ADLB_NUM_RANKS"));
    userTypes = make([]int, 2)
    userTypes[0] = 77
    userTypes[1] = 88
    rc = adlb.ADLB_Init()
    if rc != 0 {
        panic("ADLB_Init failed\n")}
    myRank = adlb.ADLB_Rank()
    numRanks = adlb.ADLB_Num_ranks()
    fmt.Printf("rank %d of %d ranks is running\n", myRank, numRanks);
    msgtosend.R = myRank + 100
    msgtosend.I = 44
    msgtosend.S = "howdy"
    msgtosend.F = 32.5
    marshMsg, err := json.Marshal(&msgtosend)
    if err != nil {
        fmt.Printf("Marshal: %v", err)}
    targetRank = 0
    workType = 88
    workPrio = myRank + 100
    fmt.Printf("CALLING ADLB_PUT\n");
    rc = adlb.ADLB_Put(marshMsg, targetRank, myRank, workType, workPrio)
    fmt.Printf("PAST ADLB_PUT\n");
    if myRank == 0 {
        for i = 0; i < numRanks; i++ {
            reqTypes = make([]int, 2)
            reqTypes[0] = -1
            reqTypes[1] = -1
            fmt.Printf("tadlb1: DOING RESERVE\n")
            rc, handle = adlb.ADLB_Reserve(reqTypes)
            fmt.Printf("tadlb1: RESERVED rc %d handle %v\n", rc, handle)
            rc, marshMsg = adlb.ADLB_Get_reserved(handle)
            fmt.Printf("tadlb1: GOT MARSH rc %d marsh %v\n", rc, marshMsg)
            err = json.Unmarshal(marshMsg, &msgrecvd)
            if err != nil {
                fmt.Printf("Unmarshal: %v", err)}
            fmt.Printf("tadlb1: GOT :%v: %d %d %s %f\n",
                msgrecvd, msgrecvd.R, msgrecvd.I, msgrecvd.S, msgrecvd.F)
        }
    }
    fmt.Printf("tadlb1: CALLING ADLB_FINALIZE\n");
    rc = adlb.ADLB_Finalize()
    fmt.Printf("tadlb1: PAST ADLB_FINALIZE\n");
}

```

Figure 2 Sample application program for ADLB-Go

```

package main
import "goadlb"
import "fmt"
func main() {
    var myrank, numserv, aprintf, numtype, amserver, commsize int
    var mytypes []int
    var i int
    var putbuf []int
    var req_types []int
    var numints int
    numints = 262144
    req_types = make([]int, 4)
    putbuf = make([]int, numints)
    numserv = 1
    aprintf = 1
    numtype = 15
    mytypes = make([]int, numtype)
    for i=0; i < numtype; i++ {
        mytypes[i] = (i+1)*10
        //fmt.Printf("Type = %d\n", (i+1)*10)
    }
    goadlb.GOADLB_Init(numserv, aprintf, numtype, mytypes, &amserver, &commsize, &myrank)
    if amserver != 0 {
        goadlb.GOADLB_Server(10000000.0,0.0)
        fmt.Printf("Comm Size = %d\n", commsize)
        fmt.Printf("My Rank = %d\n", myrank)
    } else {
        var rc int
        var reqworktype int
        reqworktype = 10 * ((myrank + 2) % commsize)
        if reqworktype == 0 {
            reqworktype = 10}
        for i = 0; i < numints; i++ {
            putbuf[i] = (myrank + 1)%commsize}
        rc = goadlb.GOADLB_Put(putbuf, numints*4, -1, 0, 10*((myrank+1)%commsize), 0)
        req_types[0] = reqworktype
        req_types[1] = -1
        req_types[2] = -1
        req_types[3] = -1
        fmt.Printf("RC from GOADLB_Put = %d\n", rc)
        var work_type int
        var work_prio int
        var work_handle []int
        var work_length int
        var answer_rank int
        work_handle = make([]int, 10)
        rc = goadlb.GOADLB_Reserve(req_types, &work_type, &work_prio, work_handle,
&work_length, &answer_rank)
        if rc < 0 {
            fmt.Printf("Error on GOADLB_Reserve\n")}
        var getbuf []int
        getbuf = make([]int, work_length/4)
        rc = goadlb.GOADLB_Get_reserved(getbuf, work_handle, work_length)
        if rc > 0 {
            fmt.Printf("RC from GOADLB_Get_reserved = %d\n", rc)
            fmt.Printf("getbuf[0] = %d\n", getbuf[0])
            fmt.Printf("getbuf[1] = %d\n", getbuf[1])
            fmt.Printf("getbuf[%d] = %d\n", numints-2, getbuf[numints-2])
            fmt.Printf("getbuf[%d] = %d\n", numints-1, getbuf[numints-1])
        } else {
            fmt.Printf("Error during GOADLB_Get_reserved\n")}
    }
    goadlb.GOADLB_Finalize()
}

```

Figure 3 Sample application program for Go2ADLB

6. Conclusions and Future Work

We have developed both an ADLB-Go package and a Go2ADLB interface for ADLB. With the availability of Go2ADLB, it is now possible for programmers to access the ADLB library from within a Go program. Both packages are freely available. However, we do not recommend the ADLB-Go prototype since it has not been optimized and is not fully functional.

The primary difficulty with this project has been the rapidly changing nature of Go. As of the writing of this paper, there have been more than 40 releases of Go since December 2009 [7]. We currently have to package the appropriate version of Go with our Go2ADLB interface because each new release of Go can potentially require changes in the interface. For future work, when Go stabilizes, we plan to fix the interface one last time and then optimize it.

7. References

- [1] ADLB: Asynchronous Dynamic Load Balancing, <http://www.cs.mtsu.edu/~rbutler/adlb>
- [2] Asanovic, K., et. al., "The Landscape of Parallel Computing Research: A view from Berkeley," 2006, Technical Report UCB/EECS-2006-183. [Online]. Available: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html>
- [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>
- [4] Baugh, John P., *Go Programming*, 2010.
- [5] Department of Energy Office of Advanced Scientific Computing Research Congressional Budget 2012 www.science.energy.gov/obp/FY_12_Budget/pdf/FY_2012_ASCR_Congressional_Budget.pdf
- [6] The Go Programming Language, <http://golang.org/>.
- [7] The Go Programming Language, Release History, <http://golang.org/doc/devel/release.html>.
- [8] Lusk, Ewing L., Pieper, Steven C., Butler, Ralph M., "More Scalability, Less Pain," *SciDAC Review* 2010 <http://www.scidacreview.org/1002/html/adlb.html>
- [9] Tiobe Programming Community Index for February 2011 <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

Evaluation iterative solver for pCDR on GPU accelerator

Chih-Wei Hsieh, Sheng-Hsiu Kuo and Chau-Yi Chou

National Center for High-Performance Computing

No. 7, R&D Rd. VI Hsinchu Science Park, Hsinchu, Taiwan, R.O.C. 30076

{david.hsieh; a00mba00; b00cyc00}@nchc.narl.org.tw

Abstract - *In the past few years, the graphics processing units (GPU) has become trend in high performance computing (HPC). The newest Top500 list was showed three supercomputers contain GPU accelerator on Top10 in Nov. 2010. The role of the GPU accelerator has become more and more important for scientific computing and computational fluid dynamic (CFD) to obtain result quickly and efficiently. The GPU has become the world's top driving force behind supercomputer. It has hundreds of processor cores in parallel, large-scale operations can be split and can simultaneously load.*

In this paper we implemented a parallel CDR (pCDR) library of using CUDA. The pCDR provides an easily high-level interface for GPU programming that greatly enhance developer productivity. It was a set of code for solving a convection diffusion reaction (CDR) scalar transport equation. In this paper, we would evaluate the performance comparison of general purpose processor and GPU accelerator. As a result, the performance of pCDR-CG via CUDA C has 6 times faster than those on a sequential code in the problem size of 800×800 .

Keywords: pCDR; Poisson Equation; GPU; Multithreading.

1 Introduction

Many-Core system plays a key role on high performance computing, HPC, nowadays. The semiconductor manufacturer encounters the wall of the speed of the central processing unit, CPU, because of the thermal management. It causes these manufactures must increase cost to cooling and global warming issue. Over the past years, the GPU accelerator has become more popular. The modern GPU is not only for display but also has many programmable cores. Such as, the Tesla C1060 which has 240 stream processors and 933 GFlops in single precision.

The bi-annual ranking of the Top500 supercomputers was shown the #1 Tianhe-1A in Nov. 2010 [13]. The Tianhe-1A linpack benchmark has 4.7 PFlops of peak theoretical and

2.56 PFlops of sustained performance. The Tianhe-1A supercomputer is comprised of 7,168 servers, each equipped with two sockets using Intel's X5670 processors running at 2.93 GHz and one NVIDIA Tesla M2050 GPU co-processor.

The role of the GPU accelerator has become more and more important for scientific computing and computational fluid dynamic (CFD) to obtain result rapidly and efficiently. There were applications using GPU implemented in CFD problem. Kruger and Westermann [7] proposed a framework for the implemented of direct solvers for sparse matrices, and apply these solvers to multidimensional finite difference equations, i.e. the 2D wave equation and the incompressible Navier-Stokes equations. Goodnight, Woolley, Lewin, Luebke and Humphreys [2] presented boundary value heat and fluid flow problems using GPU. A Navier-Stokes flow solver for structured grids using GPU was presented in [5]. Hagen, Lie, and Natvig [6] presented the implantations to compressible fluid flows using GPU. C.C. Su and C.W Hsieh [9] using GPU to implemented Monte Carlo method to solve CFD problems.

The GPU is accelerator-based platforms emerge from the issues of cost and environment. Such as, General-purpose computing on graphics processing units (GPGPU) is as an accelerator on high performance computing (HPC). It could be solve many complex and large computational problems in a more efficient than on a CPU. The Compute Unified Device Architecture (CUDA) [12] is a general-purpose parallel computing architecture developed by NVIDIA; it's implemented as the extension of ISO C99 programming language. CUDA C is available for 32-bit and 64-bit operating systems -- Linux, Windows, and Mac OS.

In this paper we implemented a parallel CDR (pCDR) library of using CUDA. The pCDR provides a high-level interface for GPU programming that greatly enhances developer productivity. The pCDR library was a set of code for solving a CDR scalar transport equation [10]. Previous study [1, 3, 8], the CUDA program has efficiently results for using red-black successive over-relaxation (Red-Black SOR) algorithm. In this study, we developed conjugate gradient (CG)

algorithm [4] in pCDR library. The CG is the most popular iterative method for solving large systems of linear equations. The CG arise in many important scheme, such as finite difference and finite element methods for solving partial differential equations, computational fluid dynamic, structural analysis and circuit analysis.

In this study we compare performance of sequential, multithreads using OpenMP [11] and pCDR-CG program. In the following, Section 2 presents high accuracy CDR scheme. In Section 3, we discuss detail of pCDR library. Section 4 and 5 presents experiment results and conclusions.

2 METHODOLOGY

In this paper we consider the finite-difference solution of the scalar convection-diffusion-reaction equation.

$$u\phi_x + v\phi_y - k(\phi_{xx} + \phi_{yy}) + c\phi = f \quad (1)$$

where u and v represent the velocity components along the x and y directions, and k and c denote the diffusion coefficient and the reaction coefficient, respectively.

Assume that f was a known value. Employ its general solution for Eq. (1) as follow

$$\phi(x, y) = c_1 e^{\lambda_1 x} + c_2 e^{\lambda_2 x} + c_3 e^{\lambda_3 y} + c_4 e^{\lambda_4 y} + \frac{f}{c} \quad (2)$$

where c_{1-4} are constants. Substituting Eq. (2) into Eq. (1), we can determine λ_{1-4} as follows

$$\lambda_{1,2} = \frac{u \pm \sqrt{u^2 + 4ck}}{2k} \quad \text{and} \quad \lambda_{3,4} = \frac{v \pm \sqrt{v^2 + 4ck}}{2k} \quad (3)$$

For the CDR model equation (1), we can discrete the equation at an interior node i . The idea is to approximate all the derivative terms using the center-like scheme

$$\begin{aligned} & \left(-\frac{u}{2h} - \frac{m}{h^2} + \frac{c}{12} \right) \phi_{i-1,j} + \left(\frac{u}{2h} - \frac{m}{h^2} + \frac{c}{12} \right) \phi_{i+1,j} + 4 \left(\frac{m}{h^2} + \frac{c}{6} \right) \phi_{i,j} \\ & + \left(-\frac{v}{2h} - \frac{m}{h^2} + \frac{c}{12} \right) \phi_{i,j-1} + \left(\frac{v}{2h} - \frac{m}{h^2} + \frac{c}{12} \right) \phi_{i,j+1} = f_{i,j} \end{aligned} \quad (4)$$

where h is the uniform grid size. Given the above discrete representation of Eq. (1), the prediction quality depends solely on m in Eq. (4). By virtue of Eq. (2), we can substitute

$$\phi_{i\pm 1,j} = c_1 e^{\lambda_1(x_i \pm h)} + c_2 e^{\lambda_2(x_i \pm h)} + c_3 e^{\lambda_3 y_j} + c_4 e^{\lambda_4 y_j} + \frac{f}{c},$$

$$\phi_{i,j} = c_1 e^{\lambda_1 x_i} + c_2 e^{\lambda_2 x_i} + c_3 e^{\lambda_3 y_j} + c_4 e^{\lambda_4 y_j} + \frac{f}{c}, \text{ and}$$

$$\phi_{i,j\pm 1} = c_1 e^{\lambda_1 x_i} + c_2 e^{\lambda_2 x_i} + c_3 e^{\lambda_3(y_j \pm h)} + c_4 e^{\lambda_4(y_j \pm h)} + \frac{f}{c} \text{ into Eq.}$$

(4) to get high accuracy. Then we can derive

$$m = \frac{\left(\frac{uh}{2} \sinh \bar{\lambda}_1 \cosh \bar{\lambda}_2 + \frac{vh}{2} \sinh \bar{\lambda}_3 \cosh \bar{\lambda}_4 + \frac{ch^2}{12} (\cosh \bar{\lambda}_1 \cosh \bar{\lambda}_2 + \cosh \bar{\lambda}_3 \cosh \bar{\lambda}_4 + 10) \right)}{\cosh \bar{\lambda}_1 \cosh \bar{\lambda}_2 + \cosh \bar{\lambda}_3 \cosh \bar{\lambda}_4 - 2} \quad (5)$$

where $\bar{\lambda}_1 = \frac{uh}{2k}$, $\bar{\lambda}_2 = \sqrt{\left(\frac{uh}{2k} \right)^2 + \frac{ch^2}{k}}$, $\bar{\lambda}_3 = \frac{vh}{2k}$, and

$\bar{\lambda}_4 = \sqrt{\left(\frac{vh}{2k} \right)^2 + \frac{ch^2}{k}}$. For time stepping scheme, we consider $\phi_t = (\phi_i^{t+1} - \phi_i^t)/dt$, which yields first-order accuracy.

3 MATERIALS AND METHOD

In this paper we implemented a parallel CDR library of using CUDA. The pCDR provides an easily high-level interface for GPU programming that greatly enhances developer productivity. It was a set of code for solving a convection diffusion reaction (CDR) scalar transport equation, input given by the convection (u, v), diffusion (k), reaction (c) to solve the physical quantity (ϕ). This equation is practically important because the working equations of many cases fall into this category. For example: thermodynamics, electromagnetic and fluid mechanics problems. We implemented two iterative algorithms in the pCDR library. First iterative method is red-black successive over-relaxation (Red-Black SOR) method. Another iterative method is conjugate gradient method (CG). The CG is an iterative method which is suited for use with sparse matrices. It can be applied to sparse systems that are too large to be handled by direct methods such as the Cholesky decomposition. Such systems often arise when numerically solving partial differential equations. Figure 1 shows the computation

procedure for the GPU application. First, initialize the data and boundary conditions in Host-side. Second, the initialization CDR coefficient in GPU, then will solving iterative method using CG or Red-Black SOR. Last, the results will restore and release memory space.

The pCDR is implemented high-level interface for user to easily programming and reduce development time. It was contains host and device functions. Users just need to know the host functions and then users can quickly develop program for applications. The pCDR provide six main functions for developers, next to explain these functions:

- `pCDR_init(u, v, k, c, dx, dy, dt, width, height)`
This function is set up CDR coefficients on GPU device.
- `pCDR_free()`
Release memory on GPU device.
- `pCDR_SOR_steady(FI, f, tolerance, MAX_STEP, Relax, width, height)`
This function using Red-Black SOR to solve CDR problem to steady state.
- `pCDR_SOR_time(FI, f, dt, Time_step, width, height)`
This function using Red-Black SOR to solve CDR problem to real transient state.
- `pCDR_CG_steady(FI, f, tolerance, MAX_STEP, Relax, width, height)`
This function using Conjugate Gradient to solve CDR problem to steady state.
- `pCDR_CG_time(FI, f, dt, Time_step, width, height)`
This function using Conjugate Gradient to solve CDR problem to real transient state.

As shown in Figure 2, main program as following four phase:

1. Data initial phase, initial the coefficients and boundary conditions in this phase.
2. Initial CDR coefficients in GPU accelerator.
3. Solve CDR problem to real transient state or steady state using GPU accelerator.

Release memory on GPU accelerator.

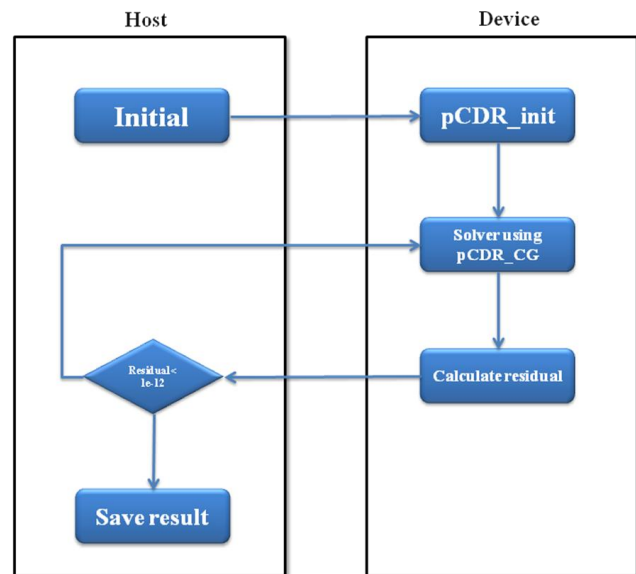


Figure 1. Flow diagram shows the library of pCDR_CG.

```
int main(void) {
    //allocate host memory to receive data
    cudaMallocHost( (void **) &FI, size );
    cudaMallocHost( (void **) &f, size );

    //set up initial data
    u, v, k, c

    //initial CDR coefficient
    pCDR_init( u, v, k, c, dx, dy, 0.0, width,height);

    //solve CG to steady state
    pCDR_CG_steady
        ( FI, f, 1.0E-12, MAX_STEP, Relax, width, height );

    //release memory
    pCDR_free();
}
```

Figure 2. The sample code of pCDR_CG

4 Result

In this paper, the researchers evaluate performance by solving the Poisson equation with exact solution. The Poisson equation is a partial differential equation of useful in computational fluid dynamic, electrostatics and mechanical engineering. In the experiment, our test bed is a Linux platform consists of two sockets INTEL Xeon X5472 Quad-Core running at 3.0 GHz, with 6 MB cache, 32 GB RAM, PCIe Gen 2.0 and a Tesla C1060, which has 240 stream processors and 933 GFlops in single precision. With CUDA-Toolkit 3.1 and NVIDIA driver 256.40 is installed in this system. As a result, we would present our study through the pCDR library to evaluate the performance comparison of general purpose processor and GPU accelerator.

The two-dimensional Poisson equation is:

$$\frac{\partial^2 \varphi}{\partial x^2} + \frac{\partial^2 \varphi}{\partial y^2} = f(x, y)$$

and with the initial and bound condition

$$\begin{cases} \varphi(x, 0) = \sin x & , 0 \leq x \leq 1 \\ \varphi(x, 1) = \sin x \cos 1 & , 0 \leq x \leq 1 \\ \varphi(0, y) = 0 & , 0 \leq y \leq 1 \\ \varphi(1, y) = \sin 1 \cos y & , 0 \leq y \leq 1 \end{cases}$$

The exact solution is

$$\varphi(x, y) = \sin x \cos y$$

In this paper we consider the finite difference solution of the scalar convection diffusion reaction equation.

In the above, the definitions of u , v , k and c are:

$$\begin{cases} u = v = c = 0 \\ k = -1 \end{cases}$$

Tables 1 and 2 show the number of convergence step of CG and Red-Black SOR for the CUDA respectively. It shows the CG has less convergence step than Red-Black SOR. We compare four different grid sizes, which are 100×100, 200×200, 400×400 and 800×800, respectively. Table 3 shows

the computations and total execution time of compare sequential, multithread and GPU accelerator. As result, we can find tradition sequential program is better than parallelism implement in problem size square of 100. From this result, it present multithreads and GPU expends more execution time of 3.7 and 4.5 times, respectively. Whereas, the parallelism performance was raised when increase problem size, as show in Figure 3.

Figure 4 show that the speed-up of CUDA-CG method running on a single GPU card, the CUDA code shows 6 times speed-up at problem size 800×800. As the results, the pCDR library has better performance in large problem size.

Table 1. The number of convergence step of Red-Black SOR.

	2000	10000	20000	40000	63000
Red-Black SOR	6.04E-05	5.08E-06	2.70E-07	7.63E-10	8.95E-13

Table 2. The number of convergence step of CG.

	1000	2000	3000	3400	3605
CG	252.534	0.0349155	4.54E-08	8.21E-11	9.82E-13

Table 3. Computation time (seconds) for different grid size.

	CG_CPU	CG_OpenMP	CG_CUDA
100×100	0.052	0.191	0.235
200×200	0.540	0.449	0.577
400×400	8.389	1.407	2.579
800×800	106.595	31.997	17.415

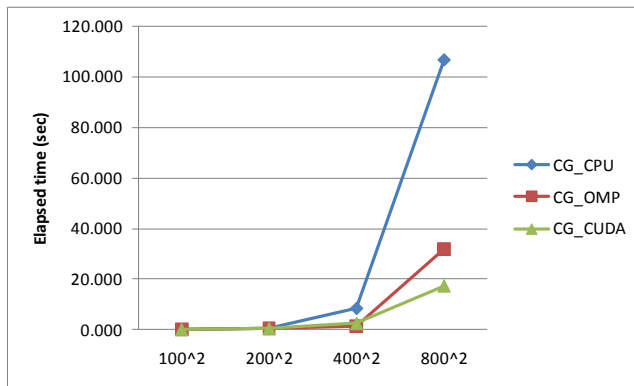


Figure 3. Computation Time (seconds) for different grid size.

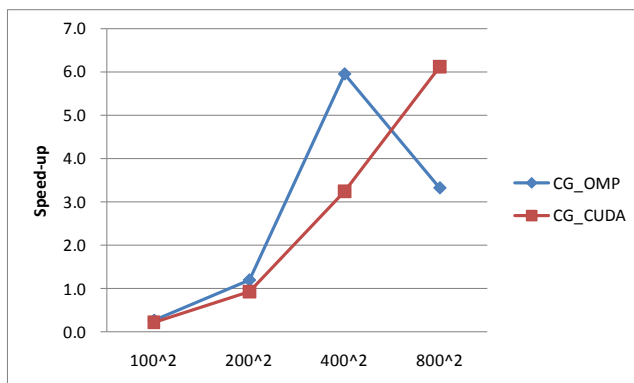


Figure 4. The performance speed-up for comparison OpenMP and CUDA

5 DISCUSSION AND CONCLUSIONS

In this paper, we propose easily and usefully application interface for GPU programming that greatly enhances developer productivity. We implement a well known iterative algorithm of conjugate gradient method for the pCDR library. As experimental results, the pCDR-CG has less convergence step than the pCDR-SOR. In addition, we evaluate elapsed time and speed-up on different grid size of 100×100 , 200×200 , 400×400 and 800×800 , respectively. As result, the sequential program shows better performance when grid size is smaller than square of 100. The multithread implemented was better performance when grid size is between 200×200 and 400×400 . The pCDR library has more efficiency performance when increase grid size. We think this study will help researcher to easily develop application using GPU accelerator.

6 References

- [1] Chau-Yi Chou, Chih-Wei Hsieh and Fang-An Kuo, "Early Evaluation on Graphic Processor based Linux Platform", HPC Asia'09 International Conference, ISBN 978-986-85228-0-0, pp. 638, Mar 2009.
- [2] N. Goodnight, C. Woolley, G. Lewin, D. Luebke and G. Humphreys, A multigrid solver for boundary value problems using programmable graphics hardware, Graphics Hardware, pp. 1–11, 2003.
- [3] Chih-Wei Hsieh, Sheng-Hsiu Kuo, Fang-An Kuo and Chau-Yi Chou, "Solving Parabolic Problems using Multithread and GPU," The 2010 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA10), Taipei, Taiwan, September 6-9, 2010.
- [4] Kenneth Hawick, Kivanc Dincer, Guy Robinson, Geoffrey Fox, "Conjugate Gradient Algorithm in Fortran90 and High Performance Fortran," NPAC Technical Report SCCS-691, April 1995.
- [5] M.J. Harris, Fast fluid dynamics simulation on the GPU, GPU Gems, pp. 637–665, 2004.
- [6] T.R. Hagen, K.A. Lie and J.R. Natvig, Solving the Euler equations on graphics processing units, Comput. Sci. – ICCS 3994, pp. 220–227, 2006.
- [7] J. Kruger and R. Westermann, Linear algebra operators for GPU implementation of numerical algorithms, ACM Trans. Graphics 22 (3), pp. 908–916, 2003.
- [8] Sheng-Hsiu Kuo, Chih-Wei Hsieh, Reui-Kuo Lin and Wen-Hann Sheu, "Solving Burgers' Equation Using Multithreading and GPU," The 2010 International Workshop on High Performance Computing Technologies and Applications (HPCTA'10), Busan, Korea, May 21-23, 2010.
- [9] C.C. Su, C.-W. Hsieh, M. R. Smith, M. C. Jermy and J.-S. Wu, "Parallel Direct Simulation Monte Carlo Computation Using CUDA on GPUs," The 27th International Symposium on Rarefied Gas Dynamics (RGD'10), Pacific Grove, California, USA, July 10-15, 2010.
- [10] W. H Sheu, S. K. Wang, R. K. Lin, "An implicit scheme for solving the convection-diffusion-reaction equation in two dimensions", Journal of Computational Physics, Vol. 164(1), pp. 123-142, 2000.
- [11] OpenMP: Open Multi-Processing, Available: <http://www.openmp.org>.
- [12] The CUDA Programming Guide, Available: http://www.nvidia.com/object/cuda_develop.html.
- [13] TOP500 Lists, Available: <http://www.top500.org>.

MOWIC: Modern Web-based Interface Toolkit for Cluster*

Daniel Cleland¹, and Chi Shen²

¹Department of Computer Science, University of Kentucky, Lexington, KY, USA

²Department of Computer Science, Kentucky State University, Frankfort, KY, USA

Abstract - *High performance computing has been developed over the years providing researchers with the most powerful computational machines available at a given time. Working conveniently and efficiently in such large complex environment for non-computer experts is one of the major concerns for computational science. The value of bridging cyberinfrastructure and application scientists is significant to advance cyberinfrastructure and their research and education. In this paper, we developed a toolkit for building web-based user interface for clusters which integrate applications to allow the users to create their own simple interface. This toolkit requires only that the user has a web browser and it can be easily used, installed with most hardware and software. The toolkit prototypes an interface that can tie together all computational programs and visualization tools in research clusters.*

Keywords: *Science gateway, User interface, Web design, Parallel program, Cluster.*

1 Introduction

Cyberinfrastructure is expanding and improving. The clusters that fulfill the needs of today's computational research get larger and more sophisticated. The National Science Foundation, which directs research and development of these resources, has outlined goals for developing sustainable and extensible HPC systems and services [1] [2]. These goals include the developments of systems that support individuals as well as groups to more easily use computation. Outreach programs work to promote awareness and provide training to increase the number of people utilizing the current infrastructure [3]. To eliminate the hurdle for common users in the usage of clusters, many science gateways in different fields have been developed as a way that enable the users associated with a common scientific discipline to use public resources or clusters through a common interface that is already configured for optimal use. However most of developed science gateways on TeraGrid resources [4] are very localized to the particular site. Moreover, the interface for much of this power remains difficult and requires expertise

in Linux shells, parallel programming or computational software packages. Hence developing a modern science gateway that can incorporate features of modern web sites, tools to view and visualize data, user authentication and connections to run the programs needed, be accessible from anywhere and be user friendly are our major goals.

Mowic, a Modern Web-based Interface toolkit for Cluster sets out to investigate the feasibility and possibility of creating a web based user interface to a high performance cluster. It provides the functionality needed by researchers to improve the human computer interaction with that cluster. This functionality includes running parallel codes, organizing the resulting datasets and allowing visualization of those datasets. This science gateway focuses on providing the end user an improved experience and productivity over existing command line interfaces.

In next section, we will outline the steps of system design, illustrate the major implementation process and present some experimental examples. Some issues related to the usage of MOWIC will be discussed in Section 3. Our future work for the next step of MOWIC development will be presented in Section 4.

2 Designing System

2.1 System Requirements

Hardware requirements for Mowic are designed to be light. It requires a web server, of which Apache was used in both CentOS as well as in Ubuntu. Due to the nature of the web interface and Apache, there is no clearly definable minimum hardware requirement for the server to host the interface. It is recommended that at least a 500Mhz computer with 128M of RAM be used as the web server. The web server must have installed MySQL and PHP [5] [6], both of which are freely available for download at www.mysql.com and www.php.net respectively. It is expected that most existing Linux servers will suffice.

* This work is supported by National Science Foundation (NSF) KY EPSCoR under Grant No. 0814194

Mowic was developed and tested on a small commodity cluster at Kentucky State University. This cluster is made up of Dell workstation computers and can be accessed at <http://ccluster.kysu.edu/mowic>. This is a Rocks cluster using CentOS, Apache, PHP, and MySQL. Rocks is an open-source software collection, including a Linux distribution - CentOS, that allows for easy creation of computational clusters [7].

2.2 Structure and Design

The main goal of MOWIC is to make the process as simple as possible to take a working HPC code, connect it to the interface, run the program however needed and visualize the resulting data in a meaningful way that can quickly be exported to a report or publication. MOWIC also intends to include ease of installation, low cost or open source components, and visual aesthetics. MOWIC system design is depicted in Figure 1.

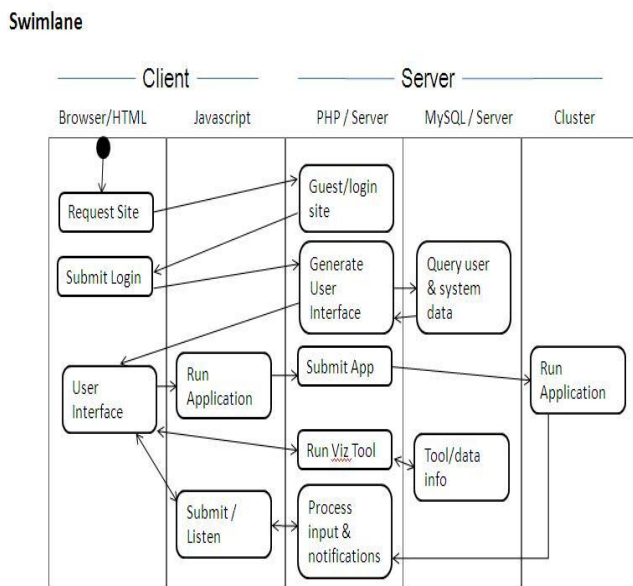


Figure 1 Interface Communications Swimlane with Ajax Calls

The processing is all text based and web calls are asynchronous and may run in the background as shown in Figure 1. The interface can run an application in the background while it waits for notification of any applications. HTML and CSS provide the basic visual aesthetics of the interface. These presentation document formats are easy to learn, develop and maintain. The W3C maintains the standards for these, most of which can be seen at the W3 schools online learning site [8]. Javascript is used to control the application communications to the server. This is done with a custom library file and jQuery. PHP provides server side control for the system. It processes requests from the browsers and sends information on to the cluster or to the

MySQL database. Information sensitive to the system is processed only in PHP so that it is never passed to the user.

The server gains access to the environment through a web user with a home directory. This user is setup as a normal cluster account, except the account is setup as a group member of the web server. This give the web server access to the data files and execution privileges to the programs found in the user directory. This will allow the site to have to make minimal change in the way they function. As long as that user can access MPI, batch schedulers, etc, then the web users will be able to as well. The system will sort out what files and programs each user has access to in that directory.

One important feature of the interface is that all the pages are loaded into the client browser upon visiting the site. The site function from then out as an application, only querying the server when the user is submitting or requesting information not loaded already. This saves bandwidth and server hits as the code is minimal due to the goal of keeping the gateway simple. Code such as the Javascript, CSS, header, footer and menu never change and should not ever be requested twice. This is avoided using Ajax calls to make http requests in the background. Ajax, asynchronous javascript and xml, is a method of using server side scripts to communicate asynchronously with a javascript based web browser application [9]. It allows for web pages to look and feel much like a standard computer application. Common Ajax applications online include Facebook, GMail, and Google Maps.

2.3 Implementation

The main menu displayed in Figure 2 reflects the simple nature of the system. It displays only five options which bring up the screen required, which are all downloaded to the client machine at the initial login of the system. A text banner is at the top of the screen with a welcome statement where the initial login box was upon entering the gateway. The Help screen displays information related to the system, the My Settings page provides the user a method to change settings and the Home page provides basic news and information.

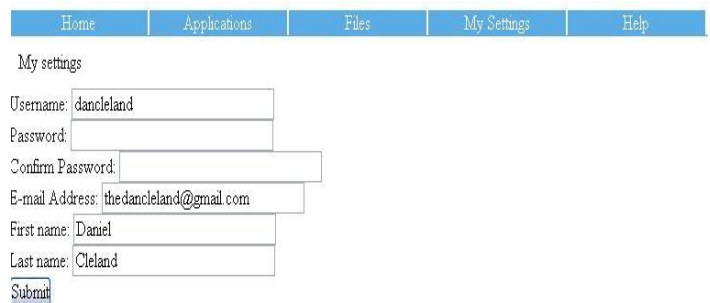


Figure 2 User Setting Screen

A Rocks Cluster with CentOS Linux was used to develop a prototype. Hand coded PHP was used to develop the first prototype. Javascript and the library jQuery was used for Ajax calls. A database was setup in MySQL to store data for the system such as file metadata and user information. The Apache server that comes with CentOS serves the content.

The Files screen in Figure 3 displays the list of files that the user has access to. These files are listed with the filename and description. The options to download the files or visualize them with a tool are also populated. Selecting "(Visualize)" results in Figure 4, a screen where the user may select what visualization tool to use on the file. These are based on the known file format of the selected file.

Interface - Cluster Gateway

Daniel Cleland, Kentucky State University

Home	Applications	Files
My list of files		
bcsttk22.mtx (download) (Visualize) - BCSSTK22: BCS Structural Engineer		
bcspwr06.mtx (download) (Visualize) - BCSPWR06: Power network pattern		
csrtest.dat (download) (Visualize) - Test CSR file sparse matrix		
out40.dat (download) (Visualize) -		

Figure 3 File Screenshot

Interface - Cluster Gateway

Daniel Cleland, Kentucky State University

Home	Applications	Files
Run Tool		
File Name:	bcspwr06.mtx	
File ID:	33	
File Type:	2	
Description:	BCSPWR06: Power network patterns Western US power network -- 1454 bus	
	<input type="radio"/> textviewer - View Contents as plain text <input type="radio"/> matrixsvg - Visualize Matrix Market Exchange MTX format matrix as an svg	
	<input type="button" value="Submit"/>	

Figure 4 Visualize with Tool Screenshot

If a user selects to visualize a data file a screen is loaded and populated with that file information. A bullet list is generated of tools that can be used to visualize that type of file. A general text view tool has access to view any file type. As for most parts of this system, the server is not needed to access this page, it is built into the downloaded application.

All such pages of the gateway act by only querying the server when absolutely necessary. Visualization tools are created to their specific task. An initial page of index.php is called in the tool folder. The tool is passed a POST variable named 'filename' that holds the name of the file that the user has requested to visualize. An 'apps' folder in the Mowic directory holds all the visualization tools. These tools must have a unique name and be loaded in to the database as an installed tool. These are the only conditions of which are then free to implement their own functions, links, and features.

A sample tool was developed as a sparse matrix viewer for matrices in MTX matrix market format [10]. This tool provides the ability to zoom in and out of the visualization. The size of the points can be expanded or contracted with the 'emphasize' and 'deemphasize' links. All of this interaction is done in browser without querying the server. This is done through the nature of SVG files as XML documents that can be modified easily with jQuery Javascript. As such this relies on the processing power of the client computer and may not be suitable for extremely large datasets.

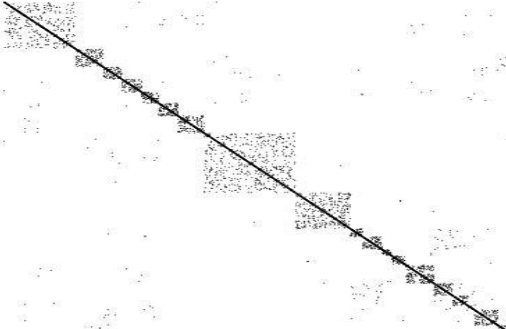
Home	Applications	Files
SVG Rendering of bcspwr06.mtx		
Zoom in Zoom Out Emphasize Deemphasize		
		

Figure 5 Sparse Matrix SVG Viewer Tool


Home	Applications	Files
SVG Rendering of bcspwr06.mtx		
Zoom in Zoom Out Emphasize Deemphasize		
		

Figure 6 Emphasized Matrix in SVG Visualization tool

In Figure 5 and Figure 6, a 1454 by 1454 sparse matrix with 3377 non-zero entries is used as an example. The total code needed to develop this particular tool was 15 lines of php/html in index.php file, 57 lines of jQuery/Javascript in scripts.js, and 52 lines of php/xml/svg in svgenerator.php. Some library Mowic system functions were accessed by linking to the global.inc.php file included with Mowic. This is an example of a visualization tool that is reusable for any matrix in that particular format that was developed in one afternoon. This could be easily modified or adapted for different formats. By changing roughly 25 lines of code in svgenerator.php another tool was developed to visualize sparse matrices in CRS format. Once the database information is entered for the tool, and permissions are setup for the user, the tool becomes usable.

2.4. Sample Parallel Code and Output Viewed

Mowic is designed to allow a specialized visualization team to be able to develop and deploy visualization software independent of the parallel code, just as in this example. The interface team or person may often provide feedback to the programmers, but has many options available to connect the interface to applications and tools. Such a tool as simply to remove repeating space characters from a text file may be useful and could easily be integrated. As the system gets larger and more robust, it begins to mimic a full desktop operating system, yet yields the computational power of the cluster behind it. The following explains the steps needed to run code on a cluster and view the resulting file with a visual tool. The particular visual tool in this case is simply a plain text viewer.

1. Open a web browser on a computer connected to the internet
2. Navigate to <http://cscluster.kysu.edu/mowic>
3. Login to Mowic with username: testuser and password: testpass
4. Click the 'Applications' menu option to list installed applications, for example, Click 'MPI_Test' to chose this simple parallel code and select any choice of number of processors to use

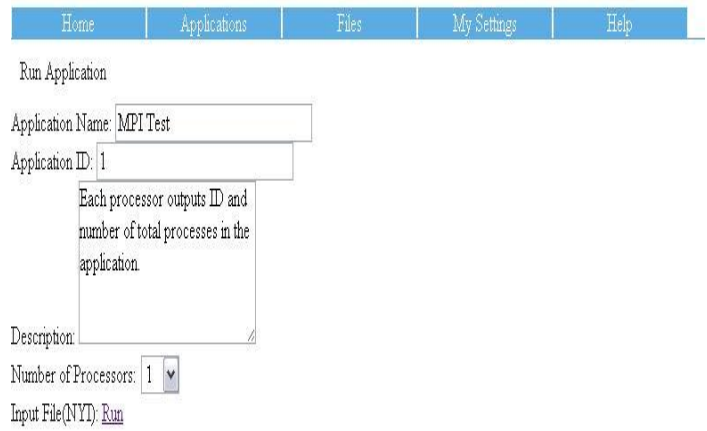
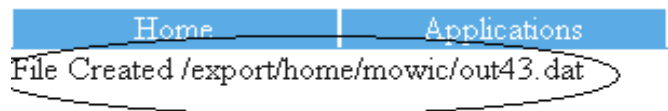


Figure 7 MPI Run Application Screen

5. When you click "Run", an output filename will be created



6. Visualize the output file by clicking the 'Files' menu option, see the screenshot on below.

My list of files

bcsstk22.mtx ([download](#)) ([Visualize](#)) - BCSSTK22: F

bcspwr06.mtx ([download](#)) ([Visualize](#)) - BCSPWR06:

csrtest.dat ([download](#)) ([Visualize](#)) - Test CSR file spa

out42.dat ([download](#)) ([Visualize](#)) -

out40.dat ([download](#)) ([Visualize](#)) -

out43.dat ([download](#)) ([Visualize](#)) -

out41.dat ([download](#)) ([Visualize](#)) -

7. After click "Visualize", if this is a text file, select the 'TextViewer' radio button

Output file

Run Tool

File Name:

File ID:

File Type:

Description:

textviewer View Contents as plain text

Submit

My processor ID is 0 of 6 total cores
 My processor ID is 5 of 6 total cores
 My processor ID is 2 of 6 total cores
 My processor ID is 3 of 6 total cores
 My processor ID is 4 of 6 total cores
 My processor ID is 1 of 6 total cores

If you chose a matrix file and want to visualize this matrix, click "bcstkt22.mtx" file at step 5. Then chose "matrixsvg" and display it

Engineering Matrices (eigenvalue problems) Textile loom frame

Description:

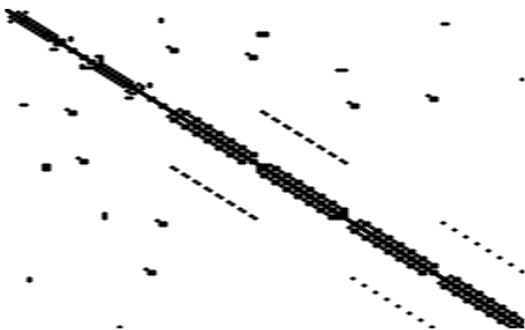
textviewer - View Contents as plain text

matrixsvg Visualize Matrix Market Exchange

Submit

SVG Rendering of bcstkt22.mtx

[Zoom in](#) | [Zoom Out](#) | [Emphasize](#) | [Deemphasize](#)



3 HOW TO USE MOWIC

The installation of Mowic is minimal. A Mowic user is created and assigned to the web server group. The web folder is copied into a web accessible directory. The SQL file included is run in a newly created MySQL database. The three system settings are adjusted in the MySQL database to reflect the location of the Mowic user's home directory and data directory to be used. A first administrator user must be manually inserted into the database. The password field should be processed through the md5 function. A config.php file is used to code in the username and password needed to connect to the MySQL server. This can be edited in any text editor. These steps complete installation.

3.1 MOWIC Administration

An administrator for Mowic should be familiar with the web server and OS of the cluster. MySQL or general SQL knowledge is helpful during the installation process but not absolutely necessary. This requirement can be further mitigated with the use of PHPMyAdmin which can easily be installed to provide a visual look into the database if needed. The administrator should be a web programmer capable of working with PHP, XHTML, and Javascript. Any other experience is useful but not critical. Custom tool development can take advantage of talents with silverlight, flash, java applets, jQuery, HTML5 canvas, etc. Computational Scientists could continue to develop programs as normal. The tool developer should have no problem quickly connecting applications and tools to the interface.

3.2 M MOWIC Application Development

Application development for cluster with a Mowic interface may function as they regularly do. Output should be directed to stdout, standard output stream, which will be redirected to an output file. Parameters should be clearly defined and setup in the installation of the application by the administrator before using. The program currently must write to one output file only. If multiple files are needed to be written to, at the current time, a delimiter can be used and the file split up after completion by a separate program. The current version of Mowic does not support batch schedulers, and is currently targeting small clusters using mpirun. This feature is planned for the next version of the interface.

3.3 P MOWIC Visualization Tools

Visualization tools are built as a site within a site. These are installed in the apps directory of Mowic in their individual folders. The folder name becomes the unique app name of the tool. The tool will be embedded in a frame within the interface so only requires navigation related to the tool. As many pages and files as needed may be stored within. This may use standard html design, Ajax [9], with any server side programming required. It may make command line calls to anything in the Mowic user's directory. This allows for batch

files to be written and integrated within them to visualize with Matlab, gnuplot, ParaView, or other tools. Visualizations are left to the designer's creativity and may include HTML5 canvas visualizations, flash, silverlight, etc. The filename to be visualized will be passed in a POST argument named 'filename'. The PHP relational include 'include(../system/global.inc.php);' [11] [12] is used to access Mowic library functions which will be provided upon request.

4 FURTHER DEVELOPMENTS

Many features are in the pipeline for further development of Mowic. The nature of such a tool to change cluster interactions so dramatically will require many updates to ensure maximum usefulness and compatibility. Batch schedulers will need to be usable with Mowic. The system may be extended also to allow for users to access their own directories for those with cluster accounts already.

Many, many visualization tools will need to be developed. An online help system including documentation and video demonstrations should be created online to support the interface. APIs could be created to collect data, allowing remote sensors to feed data directly into the system via the web. Users could have sharing tools. Chat and collaboration tools could be implemented. A mobile version of the site could be created allowing users to view the status of jobs, email the results to collaborators, etc.

5 References

- [1] Beniof, Marc R. and Lazowska, Edward D. *Computational Science: Ensuring America's Competitiveness. Networking and Information Technology Research and Development*. 2005.
- [2] Software Infrastructure for Sustained Innovation . *National Science Foundation*. [Online] National Science Foundation. http://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503489&org=NSF&sel_org=XCUT&from=fund.
- [3] HUBzero Platform for Scientific Collaboration. [Online] <http://www.hubzero.org>.
- [4] Wang, Shaowen, Wilkins-Diehr, Nancy and Martin, Stuart. Enabling Geosciences Gateways to Cyberinfrastructure. *Computers & Geosciences*. December 12, 2009, pp. 2283-2294.
- [5] MySQL Homepage. *MySQL*. [Online] <http://www.mysql.com>.
- [6] PHP Manual. [Online] <http://php.net/manual/en/index.php>.
- [7] Rocks Clusters Documentation. *Rocks Clusters*. [Online] www.rocksclusters.org.
- [8] W3Schools Online Tutorials. [Online] <http://www.w3schools.com>.
- [9] Holdener III, Anthony T. *Ajax The Definitive Guide*. Sebastopol, CA : O'Reilly Media, Inc., 2008.
- [10] *Netlib Repository at UTK and ORNL*. [Online] <http://www.netlib.org/>.
- [11] Malan, David J. *Harvard E-75 Online: Building Dynamic Website*. s.l. : <http://cs75.tv/2009/fall/>, 2009.
- [12] Williams, Hugh E. and Lane, David. *Web Database Applications with PHP and MySQL*. Sebastopol. CA : O'Reilly Media, Inc., 2004.

A Hybrid Software Framework for the GPU Acceleration of Multi-Threaded Monte Carlo Applications

Joo Hong Lee, Mark T. Jones and Paul E. Plassmann

Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Virginia 24061

Abstract – Monte Carlo simulations are extensively used in wide of application areas. Although the basic framework of these is simple, they can be extremely computationally intensive. In this paper we present a software framework partitions a generic Monte Carlo simulation into two asynchronous parts: (a) a threaded, GPU-accelerated pseudo-random number generator (or producer), and (b) a multi-threaded Monte Carlo application (or consumer). The advantage of this approach is that this software framework can be directly used in most any Monte Carlo application without requiring application-specific programming of the GPU. We present an analysis of the performance of this software framework. Finally, we compare this analysis to experimental results obtained from our implementation of this software framework.

Keywords: Parallel Monte Carlo algorithms, GPU acceleration, Hybrid algorithms, Scientific computing

1. Introduction

Recently there has been a tremendous amount of interest in using Graphics Processing Units (GPUs) to perform computationally intensive tasks traditionally performed by CPUs. This interest is motivated by the raw peak performance numbers available on current GPUs—these peak performance numbers can 1,000 times that of the associated CPU. However, there are a number of problems in achieving a high percentage of this peak performance. These problems include the parallelization of complex algorithms into large numbers of lightweight threads, the overhead of copying data between CPU and GPU memories, and the difficulties of developing GPU programs.

GPU programming has been done with the CUDA API of NVIDIA [1] or Brook+ of AMD [2]. However, each programming API is compatible with only its own hardware. Recently, the portable API OpenCL [3] has been developed for GPUs and as a result, the prospect for an over-arching portable approach for a hybrid-computing model has begun to get more attention [4]. The idea behind the portable, hybrid approach is the use of multiple threads to exploit the multiple cores on GPU.

An example of where this Monte Carlo framework can be used is PathSim2, a software environment developed to simulate biological systems at the cellular level focused on Germinal Center (GCs) [5]. The goal of these simulations is to model an adaptive immune response of the human tonsil [6-7]. As this biological system represents the motion, interaction and aging of large number of agents, it requires significant computing power. In particular, the efficient usage of computing power of GPU must be coupled with the whole simulation model to increase the performance of the system simulation. The key to make the simulation faster depends on how to transfer the CPU work to the GPU side in an efficient manner. As PathSim2 requires a parallelization strategy that can speed up the simulation using GPUs, we use an OpenCL-based solver.

Programmers in Biological System Simulation (BSS) area have started to model their program working on parallel architectures since parallel architecture have appeared and shown good performance [8]. The current trend is combining shared- and distributed-memory programming models together [9-10]. The parallel-programming techniques have evolved to take advantage of the emergence of multi-core, distributed memory computer architectures with GPUs [11]. The parallelization approach developing for PathSim2 also follows parallelization strategies in current BSS trend.

The remainder of this paper is structured as follows. In section 2 we present an overview of the simulation model and our approach to parallelization. A theoretical analysis of the performance of this proposed approach is described in section 3. In section 4 we compare experimental results of the performance of the simulation framework with the theoretical model. We present our conclusion in section 5.

2. Simulation Model Overview

PathSim2 is a software framework that simulates the motion and interaction of biological agents within a discretized three-dimensional spatial region. In the discretization of the physical volume, we refer to the discretized sub-volumes as elements and the collection of elements that make up the physical volume as the computational mesh. Thus, the movement of cells in a tissue is modeled to the movement of agents between

neighboring elements in this computational mesh. A simplified three-dimensional illustration of the elements and agents is displayed in the top image in Figure 1. In the top image of Figure 1, an element is indicated as E_k , the internal work of interaction and movement of agents is indicated as W_i^k and the summation of internal work of each agent is S_k . In the bottom image of Figure 1, we show a cropped, two-dimensional cross-section from a PathSim2 simulation [12]. This image shows a rendering of cells (agents) and elements (indicated by the size of the colored squares).

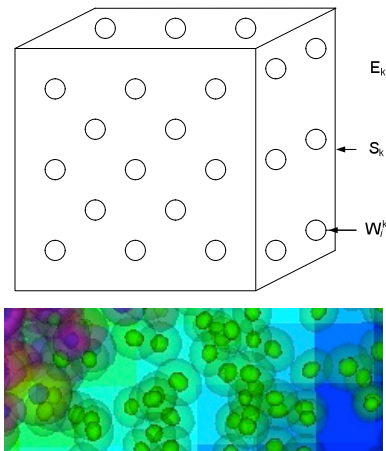


Figure 1. (Above) A simplified three-dimensional model of agents with elements, E_k : Element, W_i^k : Internal work of interaction and movement of agents, S_k : Summation of internal work of each agent; (Below) A close up of a two-dimensional cross-section from a PathSim2 simulation showing cells (agents) and elements (indicated by the colored squares [12]).

Since these element-based calculations are independent, they can be executed in separate threads. Each separate thread is distributed to each core on CPU then the computational part is calculated on main memory. However, this computational part also requires data and it can be transferred from GPU. The required data is generated on GPU then copied back to main memory for computing. Above process is described in the memory model of CPU and GPU in Figure 2.

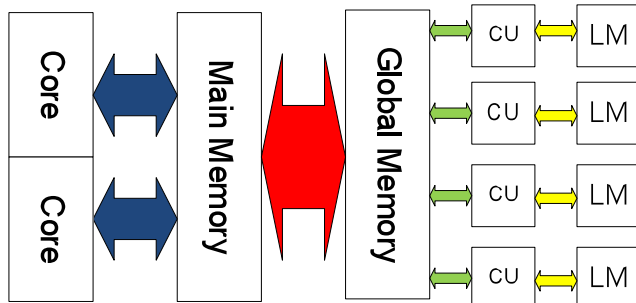


Figure 2. A Memory Model showing the CPU and GPU Architectures. CU: Compute Unit, LM: Local Memory.

Many of the computationally intensive element-based computations can be allocated to multiple threads and be distributed to multiple CPU cores. Certain other parts of the calculation, for example randomly generated agents, can be accomplished on the GPU. This allocation tasks to a multi-core system with an attached GPU is illustrated in Figure 3.

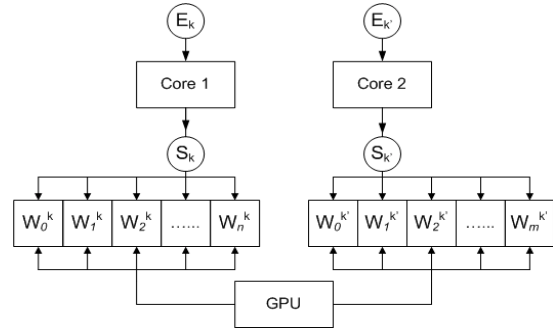


Figure 3. The assignment of element workloads to multiple cores and the GPU.

In Figure 3, the element sets assigned to the two cores are indicated as E_k and $E_{k'}$. The sets of agents within the element sets are denoted by S_k and $S_{k'}$. For the agents in these sets, the updating of the individual states (this could, for example, involve the solutions of ODEs) is represented by the work $W_0^k, W_1^k, \dots, W_n^k$ and $W_0^{k'}, W_1^{k'}, \dots, W_m^{k'}$. These work sets must be coordinated through shared memory. Then the individual work tasks are accomplished in parallel on the multiple stream processors on the GPU.

To do this, a “managing” thread can generate a new random number block as needed. Monte Carlo threads access memory blocks that are ready to be used, and when a block is used up, the managing thread swaps the memory block to another full memory blocks. Empty memory blocks are filled again on the GPU by calls from the managing thread. This overall process is illustrated in Figure 4.

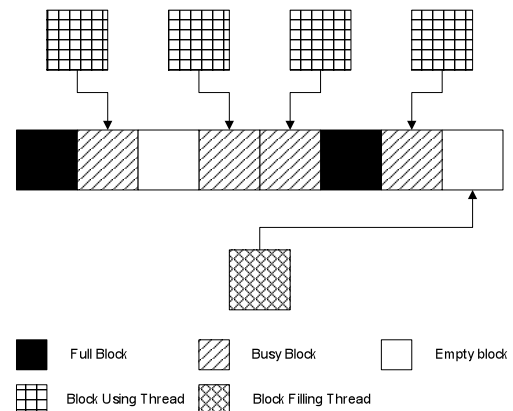


Figure 4. An illustration of how the random number blocks are managed between the GPU managing thread and the Monte Carlo application threads. The thread that manages GPU kernel fills one memory block at a time, while multiple Monte Carlo threads have access to other memory blocks that have been previously filled by the GPU managing thread.

3. Theoretical Analysis

In this section we develop a theoretical model to analyze the performance of our GPU-based pseudo-random number generation framework. For this analysis, we consider only the thread that manages the GPU kernel that is used to compute the blocks of pseudo-random numbers. We assume that the computation time is not constrained by the time required by the Monte Carlo threads that consume the numbers generated by the GPU thread. In this case, the time required by the framework is completely limited by the time required to compute the pseudo-random numbers on the GPU.

3.1. GPU Kernel Code

To develop an analysis for the computational time required to generate a block of pseudo-random numbers by the GPU thread it is necessary to examine the GPU architecture and GPU kernel code in some detail. An overview of the key section of the GPU thread code that calls the GPU kernel is shown in Figure 5. In Figure 6 we give a high-level view of the OpenCL kernel code that is executed by each thread on the GPU.

```
// Write the pseudo-random number state tables to the GPU memory
queue.enqueueWriteBuffer(PRN_Tab, PRN_Tab_Size, GPU_PRN_Tab);

// Set kernel arguments
Kernel_PRN.setArg(0, KernelCycles);
Kernel_PRN.setArg(1, GPU_PRN_Tab);
Kernel_PRN.setArg(2, GPU_PRNs);

// Iterate by calling the GPU kernel a number of times to compute
// an entire block of pseudo-random numbers
for(int Iter=0; Iter<NumIterations; Iter++){

    // Execute the pseudo-random number kernel on the GPU
    queue.enqueueNDRRangeKernel(Kernel_PRN);

    // Read back a partial block of newly computed pseudo-random numbers
    queue.enqueueReadBuffer(&PRNs[i*offset], PRNs_Size, GPU_PRNs);
}

// Read back the pseudo-random number state tables
queue.enqueueReadBuffer(PRN_Tab, PRN_Tab_Size, GPU_PRN_Tab);
```

Figure 5. A simplified overview of the OpenCL calls used to compute a block of pseudo-random numbers on the GPU. The variables PRN_Tab and PRNs are pointers to arrays in the CPU main memory for the pseudo-random number state tables and the buffer of pseudo-random numbers. The variables GPU_PRN_Tab and GPU_PRNs are pointers to memory on the GPU.

From the OpenCL pseudo-code shown in Figure 6, one can see that the required computation time is comprised of the time required to complete three types of tasks. First, data must be written from the CPU memory to the GPU memory. This task is accomplished by calling the OpenCL function *queue.enqueueWriteBuffer*. Second, the OpenCL kernel must be run on the GPU. This task is accomplished by the OpenCL function *queue.enqueueNDRRangeKernel*. Note that the kernel arguments are set by the OpenCL calls

to the function *Kernel_PRN.setArg*. And, third, data must be read back from the GPU memory to the CPU memory. This task is accomplished by the OpenCL function call *queue.enqueueReadBuffer*.

```
__kernel void KernelPRN( global KernelCycles,
                        global float *PRN_Tab,
                        global float *PRNs)
{
    // Number of workgroup
    int gid = get_global_id(0);

    // Number of workgroup size
    int global_size = get_global_size(0);

    // four-vector used as a return argument for the pseudo-random number
    // generator
    float4 randomnr = 0;

    // Generate pseudo-random numbers and then copy to GPU PRN buffer
    for ( int i = 0; i < KernelCycles; i += 4){
        randomnr = random_generator();
        PRNs[gid + (i+0) * global_size] = randomnr.x;
        PRNs[gid + (i+1) * global_size] = randomnr.y;
        PRNs[gid + (i+2) * global_size] = randomnr.z;
        PRNs[gid + (i+3) * global_size] = randomnr.w;
    }
}
```

Figure 6. A high-level view of the kernel code run on the GPU. The arguments passed to the kernel include the number of “kernel cycles” and pointers to the pseudo-random number generator state tables (input and output) and to the pseudo-random number block (output). Each GPU thread uses its workgroup number and size to write the numbers it computes to the correct GPU memory location in the PRNs buffer.

3.2. Speedup

We first consider the problem of modeling the time to read and write data between the CPU memory and the GPU memory. As has been noted elsewhere [13], a linear model can accurately represent the time required to transfer data between these memories as a function of the amount of data transferred. In Figure 7, we show experimental results for the measured transfer times for both writing from the CPU memory to the GPU memory and reading from the GPU memory to the CPU memory. Note that the linear approximations differ slightly.

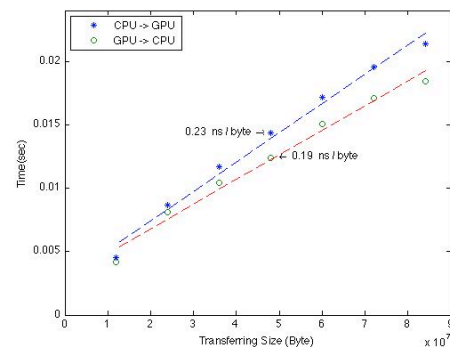


Figure 7. Data transferring time between the CPU and the GPU as a function of the number of bytes transferred. Note the different transfer rates to and from the GPU.

We use the following expression to model the time, $T_{CPU \rightarrow GPU}(m)$, to write m bytes of data from the CPU to the GPU as

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

where t_s is a “start up” time for the write, and t_{CG} is the incremental time required to write each additional byte of data. Based on a least squares fit to experimental results from a simple program that writes data between the CPU and the GPU (as shown in Figure 7), values for these constants were obtained and are presented in Table 1.

Likewise, we can use a linear model for the time, $T_{GPU \rightarrow CPU}(m)$, required to read m bytes of data from the GPU to the CPU. Thus, we use the expression

$$T_{GPU \rightarrow CPU}(m) = t_{GC} m + t_s, \quad (2)$$

where t_{GC} is the incremental time required to read each additional byte of data. Note that t_s for both the write and read are nearly equal, so we model them as the same. The incremental times for the read and write are different enough that we use two different constants as shown in Table 1.

Table 1. Constants for t_s , t_{CG} and t_{GC} as computed for the machine architecture used for the experimental tests.

Constant	Time
t_s	2.9ms/copy
t_{CG}	0.23ns/byte
t_{GC}	0.19ns/byte

On the first line of the program outline given in Figure 5, the data written to the GPU are the pseudo-random number state tables. We use n_{tab} to represent the number of bytes comprising one of the number state tables. We require a unique state table for each independent pseudo-random number generator thread that we run on the GPU. The number of GPU threads is the number of work items, n_{wi} . The number of work group items is the product of the number of work groups, n_{wg} , and the work group size, n_{wgs} . Thus, the time required to write the state tables to the GPU is given by the expression

$$T_{seedUp} = n_{wi} n_{tab} t_{CG} + t_s. \quad (3)$$

The time required to read the state tables back from the GPU (the last line of the program segment in Figure 5) is given by the expression

$$T_{seedDown} = n_{wi} n_{tab} t_{GC} + t_s. \quad (4)$$

In addition, to reading and writing the state tables, we also need to read the pseudo-random numbers generated on the GPU back to the CPU. Let n_k be the number of kernel cycles (the number of iterations in the loop in Figure 6), then the time required to read these numbers back to the CPU memory would be given by the expression

$$T_{numDown} = n_k n_{wi} t_{GC} + t_s. \quad (5)$$

The number of threads that can execute concurrently on the GPU is limited by the number of available stream processing units. However, the architecture of these stream processing units is important to account for. The stream processing units are organized into compute units, and the number of threads that are assigned to each compute unit is given by the work group size. For example, for the Radeon HD 5750 used for these experiments, the number of stream processing units per compute unit is 80. Thus, the work group size used must be at least as large as the number of stream processing units per compute unit. Overall this GPU has 9 compute units for a total of 720 stream processing units.

The effect of the GPU architecture is illustrated in Figure 8. In this figure the number of kernel cycles is fixed at 10,000; we then measure the time it takes for the kernel to execute (the call to `queue.enqueueNDRRangeKernel` in Figure 5). The measured times are shown as the green ‘+’ symbols in this graph. As we vary the size of the work group, we can observe the effect of having a limited number of stream processing units within a compute unit on which to schedule threads to execute. As the work group size increases beyond multiples of 80 (e.g., 80, 160, and 240) we observe discrete jumps in the measured times. We denote the number of stream processing units per compute unit as p_{wgs} . By examining experimental results (similar to those in Figure 7), we empirically determined that the execution time depends on two terms, a “kernel start up time” T_s and a “kernel execute time” which we denote by T_e . The first term can be modeled as

$$T_s = a n_k + b, \quad (6)$$

where a is an incremental rate, $200ns/kernel\text{-}cyle$, and b is fixed setup time, $0.9ms$. The second term can be modeled as

$$T_e = t_c^{GPU} \lceil n_{wgs} / p_{wgs} \rceil, \quad (7)$$

where t_c^{GPU} was measured to be $160ns$. Using this model for the execution time, we obtain the black ‘*’ points shown in Figure 8. The total time for the kernel to execute can then be modeled as

$$T_k = T_e + T_s. \quad (8)$$

To use the memory available on the GPU efficiently, the program given in Figure 5 iteratively generates random number blocks and reads these values back to the CPU as they are generated. The random number state tables only have to be copied to and read from the GPU outside of this iteration loop. We denote the number of iterations for this loop as n_i . We can then combine all the terms in our model to obtain an overall model for the time for the program given in Figure 5 to execute as

$$T_{GPU} = T_{seedUp} + T_{seedDown} + n_i (T_k + T_{numDown}). \quad (9)$$

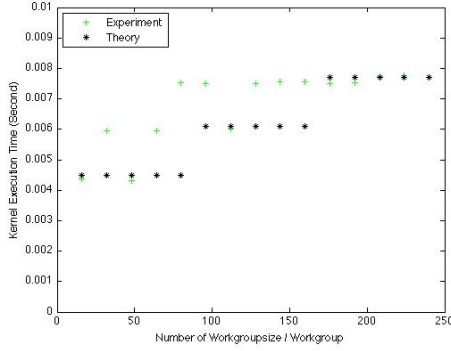


Figure 8. The time measured for the kernel to execute as a function of the work group size. For this data we fixed the number of work groups to one. The experimentally measured data is shown as the green ‘+’ points, the modeled times (as explained in the text) is shown as black ‘*’ points in this graph.

We can compare the time required to generate the pseudo-random numbers on the GPU to the time to generate these numbers using the CPU. If we denote the pseudo-random number generating rate on the CPU by t_c^{CPU} , the total required time on the CPU would be

$$T_{CPU} = n_k n_{wi} n_i t_c^{CPU}. \quad (10)$$

We can compute a speedup for using the GPU relative to using the CPU by taking the ratio of these times as

$$S = T_{CPU} / T_{GPU}. \quad (11)$$

In the subsequent section of our paper we present experimental results and compare these results to the above model. One quick observation from the model is the relatively large start up cost for reading and writing data to the GPU. It is clear that in order to amortize this start up cost, a relatively large block of pseudo-random numbers must be generated at each iteration in order to have any chance of obtaining a good speed up.

4. Experimental Results

4.1. GPU Speedup Measurement

In measuring the performance of our model, we generate a fixed number of 245,760,000 pseudo-random numbers using work group sizes of 80, 160, and 240. For the GPU we are using (a Radeon HD 5750), one compute unit consists of 80 processing elements. Thus, we increase the work group size in increments proportional to this number to allow threads to be scheduled efficiently on the GPU. The time to generate random numbers using the GPU, as compared to the CPU, is presented as a speed up in Figure 9. The experimentally measured values for t_c^{CPU} is $65ns/number$ and for t_c^{GPU} is $160ns/number$. These experimental values are used in the theoretical model presented in the previous section, and are shown in Figure 9. We also include the model where we limit the number

of compute units to 9 (and as a result the number of stream processing units to 720) and this model is also shown as the solid curves in the figure. Note that these results agree well with the theoretical model that includes the limited number of processing units available on the GPU. It is interesting to note the weak dependence of the speedup with respect to the work group size (the difference between the three curves shown in the graph). This weak dependence is due to the kernel ‘‘set up time’’ T_s .

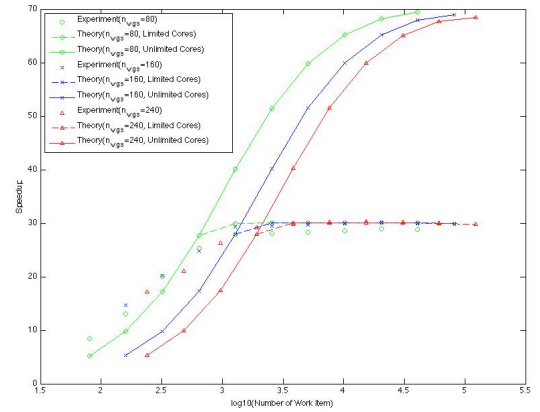


Figure 9. Speedup plots comparing the GPU execution time to the CPU execution time. Three different work group sizes (80, 160 and 240) are used. The number of work groups is increased from 1 to 512 in order to vary the number of work items.

To observe the performance improvement for using the GPU to generate the pseudo-random numbers in a simple Monte Carlo application, we consider a numerical integration scheme to estimate the value of π . The overall software framework is illustrated in Figure 4. The framework has a managing thread that fills empty memory blocks with pseudo-random numbers. The threads that use numbers for the Monte Carlo application access these full memory blocks via a shared-memory producer/consumer implementation. The pseudo-random numbers can be generated either on the CPU or on the GPU. When using the CPU, the RANLUX numbers are generated by routines from the GNU scientific library [14]. When using the GPU, memory blocks are filled with the method described in Figures 5 and 6. The simulation times for the CPU and the GPU are compared, and the resulting speedup is shown in Figure 10. This graph shows that generating the pseudo-random numbers using the GPU makes the Monte Carlo application run significantly faster when compared to using the CPU.

4.2. Randomness Check

Before using generated random numbers, we need to know that the random numbers are statistically independent. The standard way to check the randomness is as follows. First, we subdivide the samples into some

number of independent subsamples and obtain each subsample mean. Then, the standard deviation of for these subsample means should be decreasing as we increase the number of samples (a result of Central Limit Theorem). This approach can be used as a sanity check for the pseudo-random numbers generated on the independent threads on the GPU. In Figure 11, we present the measured standard deviation of pseudo-random numbers generated using the CPU and the GPU. As expected, the standard deviation decreases as the square root of the number of samples with increasing numbers of samples for both cases.

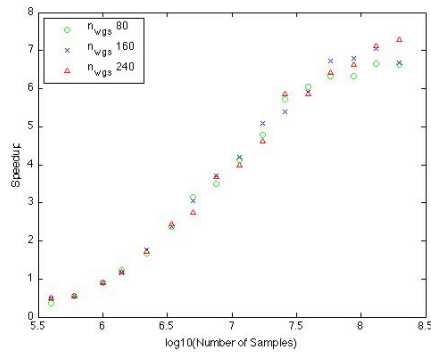


Figure 10. Speedup of a simple Monte Carlo simulation using the GPU acceleration scheme with work group sizes of 80, 160 and 240.

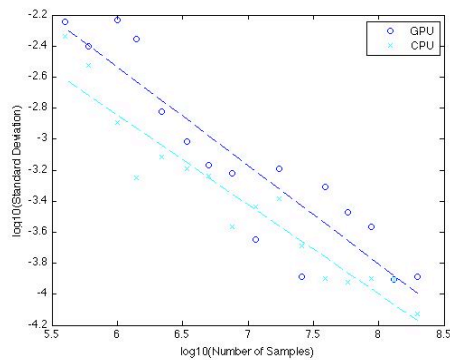


Figure 11. The standard deviation of the means computed for the Monte Carlo application as a function of the number of samples used to compute the means. As expected from the Central Limit Theorem, these standard deviations should decrease as the square root of the number of samples (the dashed lines in the figure).

As a more rigorous test, we also used the empirical tests of TestU01 to check the theoretical quality of our RANLUX pseudo-random numbers [15]. This package contains three sets of test batteries. The tests are SmallCrush, Crush, and BigCrush. These tests apply a variety of statistical tests to large sequences of random numbers. We tested the GPU implementation of RANLUX using these three tests.

One fine point in using these tests is that the GPU implementation of RANLUX has only 24 bits of resolution as it is computed in single precision. This means that when

converted to a double precision value for the tests, the additional mantissa bits in the double have to be filled with statistically independent values for the test. Once this was done, then the random numbers passed the SmallCrush battery. In the case of the Crush and BigCrush battery, except only one battery, it passes all of the tests. The tests included in SmallCrush, Crush and BigCrush respectively include 15, 144, and 160 independent statistical tests.

4.3. Monte Carlo Simulation Results

To verify the adequacy of our theoretical model, we present the estimation of π using Monte Carlo method. The estimated value of π is presented with error and mean in Figure 12. The theoretical value is within the error range of experimental result. Also with more samples, the experimental result approximates to π . More importantly, the convergence depends on the square root of the number of samples as shown in Figure 11 (which agrees with what one expects from the Central Limit Theorem).

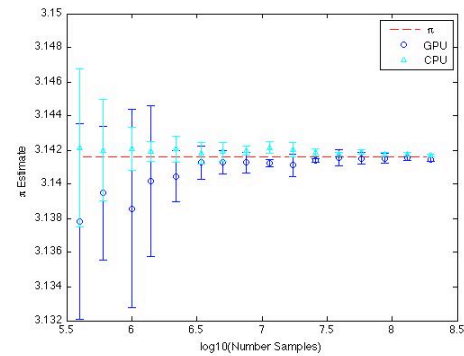


Figure 12. Convergence of the theoretical and experimental estimation of π by numerical integration with the Monte Carlo framework as a function of the number of samples used.

5. Conclusions

In this paper we have introduced a theoretical model of the efficiency of a multi-threaded Monte Carlo application framework using GPU acceleration. Experimental results are obtained by measuring the running time of the simulation framework and these running times are well explained by the theoretical analysis.

Our approach demonstrates an efficient way of mixing multi-threading with GPU acceleration. We observe that generating as much data as possible from the GPU at a time (through blocking) improves the overall simulation time relative to a CPU-based scheme. However, the time required transferring data between the CPU and GPU memories and hardware setup times ultimately limit the efficiencies of these algorithms. These limits need to be considered when considering the overall benefits possible for a GPU-accelerated software application framework.

6. ACKNOWLEDGEMENTS

This work was supported by NSF grant CCF-0728901.

7. REFERENCES

- [1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol 6, pp. 40-53, 2008.
- [2] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," *ACM Transactions on Graphics*, vol 23, pp 777-786, 2004.
- [3] J. E. Stone, D. Gohara, and S. Guochun, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol 12, pp 66-73, 2010.
- [4] G. Jost, J. Jin, D. Mey, and F. Hatay, "Comparing the OpenMP, MPI, and hybrid programming paradigm on an SMP cluster," presented at European Workshop on OpenMP(EWOMP), Germany, 2003.
- [5] "PathSim2"[online]. Available: <http://pathsim2.ece.vt.edu/>. [Accessed: Feb. 1, 2010].
- [6] "PathSim"[online]. Available: <http://pathsim.vbi.vt.edu/>. [Accessed: Feb. 1, 2010].
- [7] N. F. Polys, D. A. Bowman, C. North, R. Laubenbacher, and K. Duca, "PathSim visualizer: an Information-Rich Virtual Environment framework for systems biology," *International conference on 3D Web Technology*, pp. 7-14, 2004.
- [8] K. Stuben, "Europort-D: commercial benefits of using parallel technology," *Parallel Computing: Fundamentals, Applications and New Directions, Advances in Parallel Computing*, vol. 12, pp. 61-78, 1998.
- [9] A. Moerschell and J. D. Owens, "Distributed Texture Memory in a Multi-GPU Environment" *Computer Graphics Forum*, vol. 27, pp. 130-151, 2008.
- [10] I. Lashuk, A. Chandramowlishwaran, H. Langston, T. Nguyen, R. Sampath, A. Shringarpure, R. Vuduc, L. Ying, D. Zorin, and G. Biros, "A massively parallel adaptive fast-multipole method on heterogeneous architectures," presented at *Conference on High Performance Computing Networking, Storage and Analysis*, SC, Portland, OR, 2009.
- [11] S. Shah, and E. Gabriel, "Image computing for digital pathology," presented at *International Conference on Pattern Recognition (ICPR)*, Tampa, FL, 2008.
- [12] J. H. Lee, M. T. Jones, Paul E. Plassmann: A scalable distributed memory programming model for large-scale biological systems simulation. *International Conference on Scientific Computing (CSC)*, pp. 251-256, 2010.
- [13] O. S. Lawlor: Message passing for GPGPU clusters: cudaMPI. *Cluster Computing and Workshops*, pp. 1-8, 2009.
- [14] "GSL – GNU Scientific Library" [online]. Available: <http://www.gnu.org/software/> [Accessed: Feb.1,2011]
- [15] "TestU01"[online]. Available: <http://www.iro.umontreal.ca/~simardr/testu01/tu01.html> [Accessed: Mar.1, 2011]

Framework Construction of Energy Efficiency System of Data Center

Haiping Qu¹, Xiuwen Wang², Lu Xu³

¹ Institute of Computing Technology, Chinese Academy of Science, Beijing 100190, China

² National Computer network Emergency Response technical Team/Coordination Center of China, Beijing 100876, China

³ Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
National Engineering Laboratory for Disaster Backup and Recovery, Beijing 100876, China

Abstract - *Currently, it is widely concerned that the energy consumption has become a serious bottleneck in the development of data center. For those data centers hosting different applications, dynamic server provisioning techniques are effectiveness in turning off unnecessary to save energy. In this paper, the data center's energy efficiency is focused on the research with a framework named EADC which has two levels of energy modules (VEMS and DCMS). It builds a Virtual Environment (VE) for each service, and inside VE, VEMS can put down the energy usage by changing the status of running nodes. In the data center level, DCMS tries to satisfy the QoS of all VEs by controlling the nodes migration among VEs at regular intervals. Evolution results show that EADC makes data center achieve the balance between energy consumption and performance. And it can save a significant amount of energy consumption while meeting the QoS requirements.*

Keywords: Cluster Management; Quality of Service; Energy Efficiency

1 Introduction

Large outsourcing data centers that host many third party applications are receiving more and more attention in recent years. These applications are mutually independent and require different guarantees of QoS (Quality of Service) performance. With more and more services operating on the data center, power consumption of the data center has grown rapidly in the past few years and has become a serious bottleneck for the development of data center. Eric Schmidt, Google CEO: "It's not speed but power—low power, because data centers can consume as much electricity as a small city.[1]" According to IDC, the average energy cost of data center increases by 20% annually, while the vast majority of this growth originates from the server's power consumption. Under such circumstances, the existing strategy related to data center (with QoS guarantee for service quality serving as the main goal) requires our rethinking. Energy efficiency was introduced to the parameter list as an important factor, and by considering the relationship between performance and power

consumption comprehensively, high-performance and low-power of the data center was realized on the basis of the balance between the two, thus achieving the objective of "green data center".

As we all know, one of the major causes of energy inefficiency in data centers is the idle power wasted when servers run at low utilization. Even at a very low load, such as 10% CPU utilization, the power consumed is over 50% of the peak power [2]. We believe that switching the server state on demand and closing idle servers are effective measures to save energy. In order to achieve the balance between power consumption and performance of the data center, we designed and implemented resource management framework EADC (Energy-Aware Data Center) based on energy efficiency. This framework realizes two objectives: (1) in the service cluster level, we get the lowest energy consumption of service by controlling the service node status on demand; (2) in the data center level, by controlling the switch of service nodes between service clusters on demand, all of the service QoS on the data center is satisfied and lowest energy consumption of global resources is achieved.

The rest of the paper is organized as follows. Section 2 presents the details of system architecture and main function of software modules. Section 3 discusses our studies on the two levels of the energy module construction. Section 4 demonstrates the results of prototype experiments. Section 5 reviews the related works about energy consumption and resource placement. Finally, concluding remarks and discussion about the future work are given in Section 6.

2 Architecture of the EADC System

2.1 System Framework

The basic framework of EADC system for data center built in this study is shown in Figure 1. In Figure 1, the matrix box in solid line denotes data center, consisted of a number of physical machines; physical machines are heterogeneous, with different hardware configurations. Each physical machine is used either to build service node directly, or to build different virtual service nodes for varying application purposes through virtualization technology. The nodes operating for one application purpose compose the service cluster for this

application, which is called VE (Virtual Environment), and is denoted by the matrix box in dashed line.

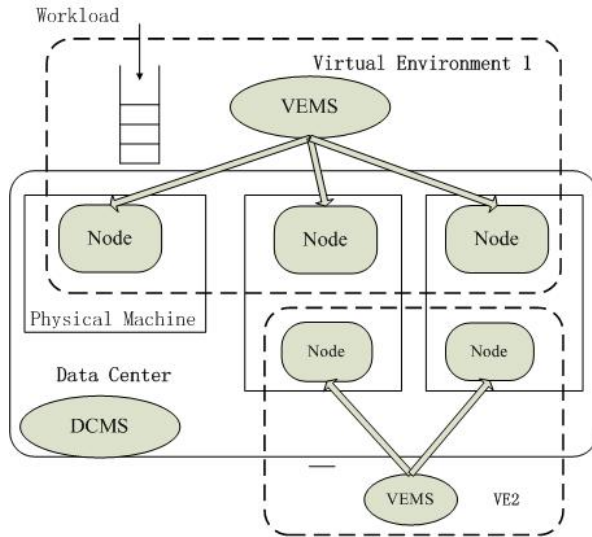


Figure1. EADC system framework

EADC system mainly consists of two parts: service cluster management system VEMS and data center management system DCMS, which are implemented on data center management server and service cluster controller, respectively. Under this system framework, we are trying to resolve the following issues:

(1) VE level (mainly focused on energy consumption): for the real-time operational service cluster, how to effectively complete service request and service node mapping in order to meet the requirement on service quality (*Request Mapping*); and how to obtain the lowest energy consumption of VE through switching the node status under the condition that good service quality is ensured (*Status Switching*).

(2) Data center level (mainly focused on performance): whether the global resource allocation of the data center meet the needs of all operating services (*Energy Assessment*); and when the service quality of a certain VE falls short of the expectation, how to conduct the fast, on-demand global scheduling of resources to meet the demand for newly emerging service resources (*Resource Scheduling*).

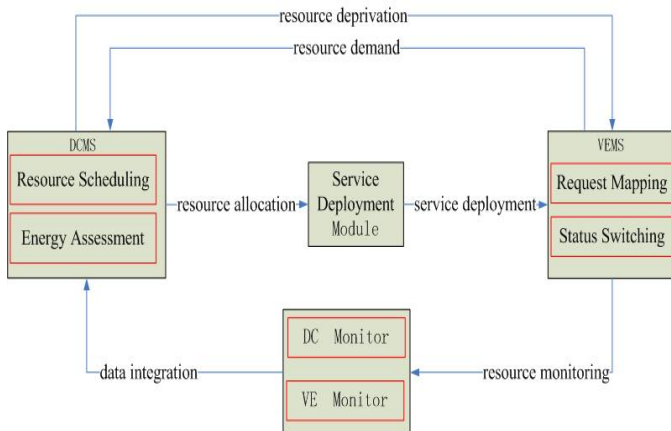


Figure2. Model Construction of EADC

Thus we constructed energy consumption model at the levels of DCMS and VEMS, and the relationship among various component modules is shown in Figure 2. It can be seen that service deployment module is the basis of our research and enabling platform. The realization of the rapid construction of VE and quick switching of service nodes among the VEs must be ensured. The service deployment system Bladmin [3] we have developed can complete service deployment of hundreds of nodes within 10 minutes. The switching of deployment nodes among the services can be achieved in seconds, which meets the research needs.

Demand and deprivation of resources is the interaction of two-level energy consumption modules:

(1) Resource demand: when unexpected service requests arise, and the service quality falls short of the expectation, VEMS sends request to DCMS, asking for the service capacity expansion;

(2) Resource deprivation: when faced with the request for capacity expansion and scarcity of idle global resources, DCMS sends deprivation request to one light-load VE for resource migration

2.2 Main Function of EADC

We can achieve the following functions under EADC framework: (1) to achieve the objective of energy saving by changing the state of the nodes (VEMS); (2) to meet the emergency resource demand of service by dynamic scheduling of the nodes (DCMS).

VEMS's main objective is to achieve the lowest energy consumption for real-time operating service cluster system on the premise of satisfying the QoS. In fact, this is to achieve the on-demand (when) and quantitative (who) switching of service nodes based on the premise of satisfying the performance. Therefore, we need to build the prediction and performance model of service cluster with energy consumption being taken into account, and to make decisions on this basis.

DCMS is actually a decision maker for global resource placement. With M VEs in the data center, it determines the M-dimensional distribution vector of resources to re-configure the owners of nodes, so that the lowest energy consumption (sufficient resources) can be obtained on the premise of satisfying QoS of all M VEs; or the minimal sum of performance dissatisfaction of M VEs (insufficient resources). We adjust the mapping relationship between the service nodes and the VE every certain time units.

Attributed to the core issues of EADC are the status switching and owner transformation of service nodes. The service node status can be divided into three types, active (ready for service), standby (service configured but in sleep status), poweroff (physical poweroff). The service node either is in a certain VE, or belongs to one idlepool (set of unallocated nodes).

The main function of EADC system is to reduce energy consumption for service under light-load, and to meet the performance for service under overload:

(1) VE under light-load: VEMS is responsible for energy conservation (selected node: active -> standby); and DCMS is first turned off (selected node: standby->poweroff) and then transferred it to idlepool if needed.

(2) VE under overload: VEMS will switch the state of selected nodes to active and if all nodes can't satisfy the service QoS then submit resource demand for new resources; DCMS transfers nodes from idlepool or other light-load VE to meet the service QoS.

3 Modeling and Utility

3.1 Problem Definition

Here we particularly focus on the online transaction class, although it applies to various kinds of applications. Thus we use the average response time as the performance metric for the QoS of a VE, and use the request arrival rate to represent the workload intensity. The parameters used throughout the paper are summarized in Table I. Energy consumption of service node in time t can be expressed as $E_t = (S_{base} + (S_{max} - S_{base}) * load) * t$ [4]. And service rate of node running constant service is defined as $\mu = c / D$.

TABLE I. PRAMETERS DEFINITION

Symbol	Explanation
c	Capacity of node
μ	Service rate of node
$load$	Utilization of node
S_{base}	Base energy consumption of node
S_{max}	Max energy consumption of node
E_t	Energy consumption of node in time t
D	Mean service demand of VE
R	Mean response time of VE
Pri	Running priority of VE
λ	Current workload intensity of VE
R	Current response time of VE
C	Capacity of VE
N	Number of nodes in data center
M	Number of VE in data center

To simplify the judgment, we normalize the factors of scheme selection as VE Capacity, which is shown in Table II. The first four parameters ($C_{min} \leq C_t \leq C_{cur} \leq C_{max}$) can be obtained from the actual system operation, and C_{t+1} is the forecast value which needs to be calculated from the forecast model and performance model.

TABLE II. PARAMETERS OF VE CAPACITY

Symbol	Explanation
C_{min}	Min capacity among active nodes (N_i in VE) return (μ_i) if i active and min
C_t	Current used VE capacity active (N_i in VE) Sum ($\mu_i * load$) if i active
C_{cur}	Current total VE capacity (N_i in VE) Sum μ_i if i active

C_{max}	Total VE capacity (N_i in VE) sum μ_i
C_{t+1}	Needed VE capacity in next time interval

We use n-order linear prediction algorithm to construct the forecast model (FM) for service load forecasting, $(trend, \lambda_{t+1}) = FM(\lambda_t, \lambda_{t-1}, \dots)$. There are two outputs of the model, trend {up/con/down} is the middle-term value, which stands for the three load trends separately: upward, constant and down; and λ_{t+1} is short-term forecast value. Let's λ_{t+1} and R is input, we get C_{t+1} through performance model.

3.2 VE Performance Model

Assuming that the current VE includes c heterogeneous service nodes, we arrange these nodes according to their unit power consumption capacity (c / S_{max}). Each request arrived completes the mapping of request to node through request transmission. In accordance with the principle of "the abler person, the busier he is", the strongest node in the queue with idle nodes is always selected to satisfy the request. If no idle node is available, the request should be in the waiting queue, and if the waiting time is too long, it may be discarded. This situation is very similar to the M / M / c models in queuing theory, except that each node capacity is heterogeneous. The building of VE performance model is based on this model; the state transition diagram is shown in Figure 3 which the service rates of the c nodes are $\mu_1 \dots \mu_c$ in descending order and n-c is the max length of waiting queue.

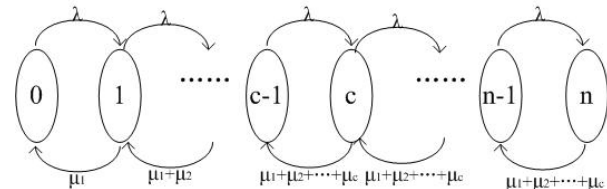


Figure3. Flow diagram of performance model

Figure 3 shows that when $n < c$, the speed of the request leaving the system increases inconstantly; but when $n \geq c$, it remains unchanged. Consequently, the differential Eq.(1) can be obtained, where P_n denotes the equilibrium probability that there are n requests in the system. Then it can be resolved by recursive method based on $\sum_{i=0}^n P_i = 1$ with state probability P_0 obtained. Thus, the average response time W_s can be calculated.

$$\begin{cases} \lambda P_{n-1} = P_n \sum_{k=1}^n \mu_k, 0 \leq n \leq c \\ \lambda P_{n-1} = P_n \sum_{k=1}^c \mu_k, n > c \end{cases} \quad (1)$$

Apart from W_s serving as an output of the model, we also need the load of each node to calculate the energy

consumption of nodes, and the load calculation of node i can be simplified as Eq.(2).

$$load_i = 1 - \sum_{k=0}^{i-1} P_k \quad (1 \leq i \leq c) \quad (2)$$

In this model, the arrival rate is required to be constant and smaller than the integrated service rate of all the servers. However, sometimes the varying load might go beyond the service capability of all servers, thus we do not make the steady-state assumption for the system. So based on [5], we modify our model by adopting a time-domain queuing model.

Let $\eta = \sum_{k=1}^c \mu_k$, if $\lambda > \eta$, all the servers will be busy serving

the incoming requests, and the queue length will keep growing or remain constant (n) during the next interval T . Denote q_0 as the measured queue length at the beginning of the next time interval, then $q(t)$ is the length of the queue at any instant t within T and q is the average queue length which get from Eq.(3) and Eq(4). We use Little's theorem to derive the average response time as $W_s = \frac{q + c}{\eta}$.

$$q(t) = \min(q_0 + (\lambda - \eta) \cdot t, n) \quad (3)$$

$$q = \frac{1}{T} \int_0^T q(t) dt \quad (4)$$

3.3 Reduction of DCMS Utility

The global resources allocation of DCMS can be actually defined as non-linear multi-objective constrained optimization problem: with the current N nodes and M VEs, then one effective resource placement corresponds to a vector of M -dimensional resource allocation: $RAV = (n_1, n_2, \dots, n_M), n_1 + \dots + n_M \leq N$. By building the DCMS utility, genetic algorithm is adopted for the solution of optimal configuration.

Since our main job is to reduce energy consumption when QoS is satisfied, and to improve the performance when QoS is violated, the two utility functions of VE are energy utility and performance utility respectively, shown as Eq.(5). When the VE is under light-load, what we care about is the lowest energy consumption of all nodes of VE. And when under overload, the important thing is the satisfaction degree about the performance of VE, it decreases very slowly when $rt < R$, once $rt > R$ the value will drop rapidly, especially when $rt = 0$, it gets the max value of 100, in which K_i and K'_i are zooming factors. By the performance model, the W_s of each VE and the load of each node can be obtained, and then we can obtain the utilities of its performance and energy consumption.

$$Ue_i = e(E_j) = \sum_{j=1}^n E_j \quad (5)$$

$$Up_i = p(rt, R) = \frac{K_i e^{(-rt+R)*K'_i}}{1 + e^{(-rt+R)*K'_i}}$$

For the global utility, the corresponding one is a piecewise function, with all VE QoS being satisfied as the judgment basis, and the process logic shown in Figure 4 is selected as the evaluation criteria for individual fitness of genetic algorithm.

```

Define status as the number of unsatisfied VE
To all placements
if (exists min status) return this
else to the placements with min status
  if (status > 0) return max(sum(Up))
  elsif(status == 0) return min(sum(Ue))

```

Figure4. Global utility for individual fitness

The reconfiguration of resources placement needs some time. Therefore the new placement should minimize adaptation time, especially the migration cost among VEs. In order to match the actual situation more precisely, our genetic algorithm takes reconfiguration delay into consideration though in an approximate way.

Assume that adding one node onto a VE from idlepool needs T_a time, while the time of removing one node from a VE to idlepool depends on the status of the node { poweroff: T_{tp} , standby: T_{rs} , active: T_{ra} }. Then the time of a new adding node ready for service may be { T_a , $T_a + T_{tp}$, $T_a + T_{rs}$, $T_a + T_{ra}$ }. T_c is used for the max time that all adding nodes will be ready. We consider the delay time when estimating the individual fitness. Compared with the current placement, the new placement of the VE that removes some nodes should take into action immediately. But it is different to the VE that adds some nodes because it needs some time for the new placement to be fulfilled. Thus, during the first T_c period in next interval, the configuration is same as the current configuration. And after time T_c , the configuration will be changed to the new placement. We use the current and new placement respectively as the input of performance model and obtain the outputs time of first T_c period and the rest period. Then the energy and performance utility of the whole interval can be derived.

4 Evaluation Experiments

4.1 Environment Construction

In order to evaluate the effectiveness of EADC framework, we particularly simulate a prototype data center comprised of 20 service nodes and three VEs, each running a class of online transaction. The nodes have four different configurations, each with 5 nodes, shown as Table III.

TABLE III. CONFIGURATIONS OF SERVICE NODES

Node	N1	N2	N3	N4
Capacity c	500	400	270	150
Smax (Watt)	250	250	150	120

We generate the workloads with 720 min based on EPA-HTTP, Day20 of WorldCup98 and Day18 of WorldCup98 [6]. It can be seen from Figure 5 that the three workloads have certain representation which shown as upward, down and constant trends separately. The workloads use Poisson request arrivals and exponentially distributed request sizes. Table IV shows the parameters of each application.

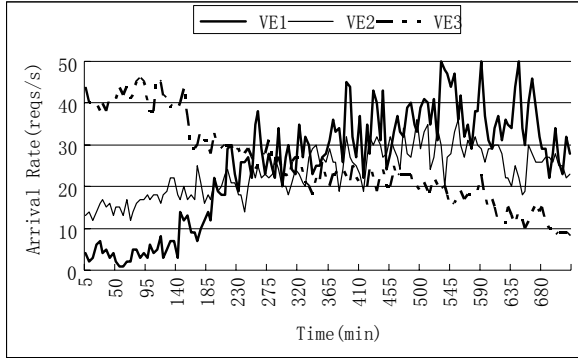


Figure5. Variation of the workload intensity

TABLE IV. APPLICATION PARAMETERS

VE	VE1	VE2	VE3
Mean service demand D	30	40	50
Mean response time R (s)	0.20	0.25	0.30
Priority Pri	0.40	0.35	0.25

4.2 Verification of Performance Model

VEMS performance model needs to be validated. In this section, we compare performance predictions by the model with measurements obtained in the experimental setting. We use 4 nodes {500,400,250,150} to construct the VE, and the arrival rate of requests (reqs/s) increase from 1 to 69. To make a clear comparison of average response time, the tests are conducted on the steady state and unsteady state of VE, as shown in Figure 6 and 7.

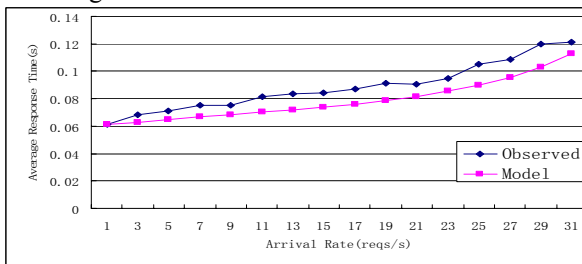


Figure6. Variation of response time of steady-state VE

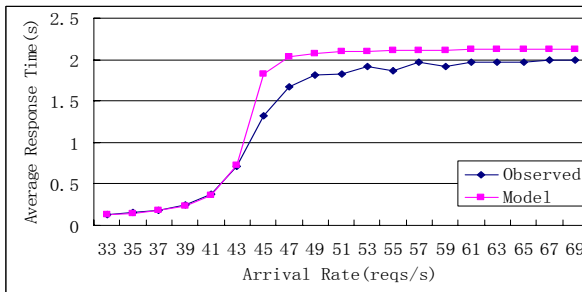


Figure7. Variation of response time of unsteady-state VE

The comparison of VE load can be seen from Figure 8. Overall, the results obtained from the analytic performance model imply that the model tracks the measurement reasonably well for its being used by VEMS and DCMS.

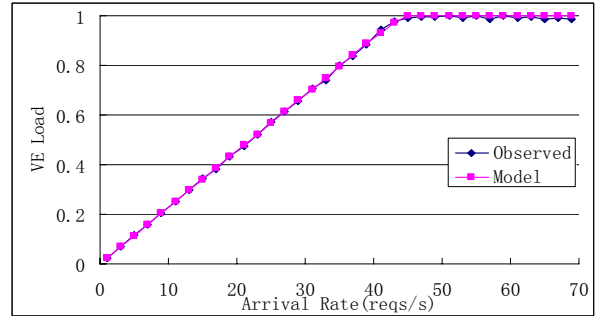


Figure8. Variation of VE load

4.3 Test of EADC Global Utility

Three different tests are conducted on resource allocation strategy to assess the decision effect of global resource placement of EADC:

(1) Static4 and Static6: 4 nodes are distributed in VE for Static4, with one node for each configuration; Static6 has 6 nodes in VE, which is based on Static4, two nodes are added in accordance with the priorities {VE1:500,400; VE2:500,270; VE3:400,270}. Static4 is the minimum configuration, which generally meets the workload needs for about half the time, while the configuration of Static6 can basically meet three VE's QoS requirement full-time.

(2) EADC: having the same initial configuration with Static4. For every 5 minutes, VEMS implements the node status switching according to the load changes; for every 30 minutes, DCMS implements re-configuration of nodes based on utility evaluation.

Figure 9 and Figure 10 are a time variation curve of global performance utility and global energy utility. It can be seen that at the same time when EADC achieved the performance consistent with Static6, its energy consumption is roughly equivalent to that of Static4. Within a period of 720min, EADC consumes 24.857 kwh, while Static4 and Static6 consume 23.583 kwh and 31.471 kwh, respectively.

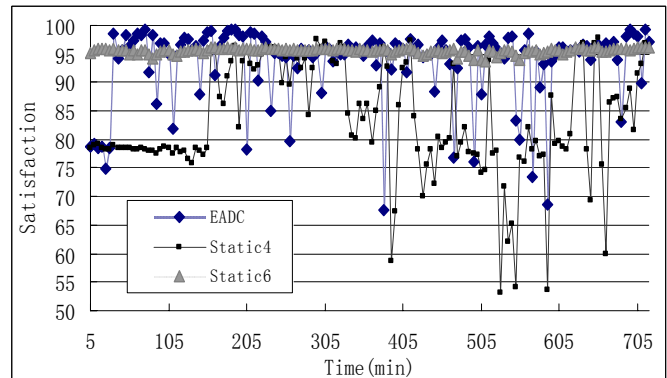


Figure9. Variation of global performance utility

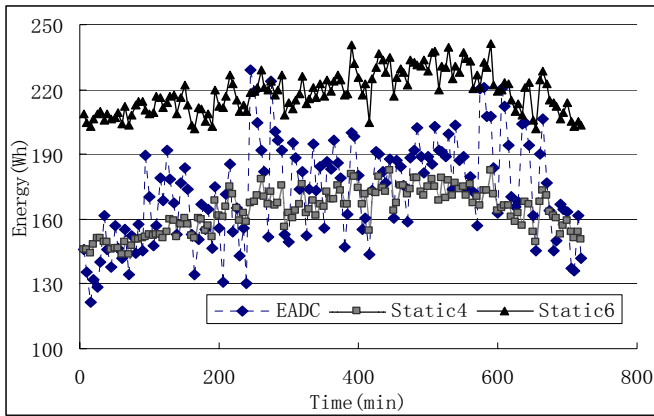


Figure10. Variation of global energy utility

Specific situation of VE1 under EADC is also checked as Figure 11 and Figure 12. Figure 11 shows the Cmax and Ccur of VE1 during the experiment, and Figure 12 shows the performance utility and energy utility of VE1. We can see that the performance satisfaction degree of VE1 does not decrease with switching of the status of nodes.

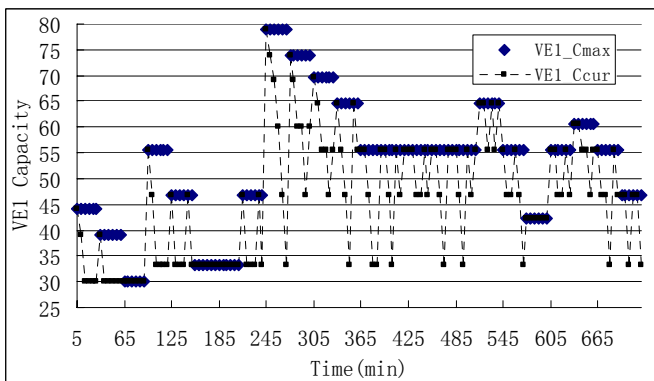


Figure11. Allocated capacity of VE1

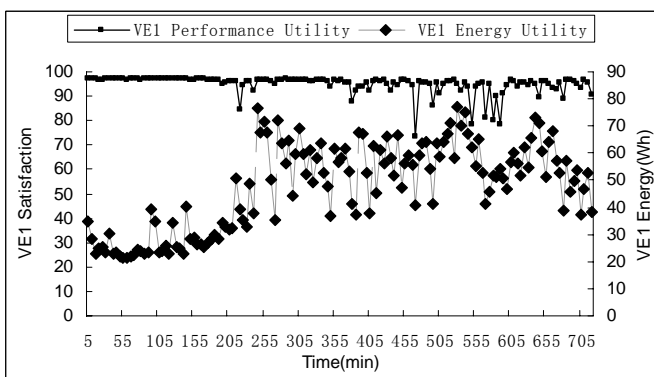


Figure12. Local utility of VE1

Experimental results show that the EADC achieves the optimized trade-off between its service performance and energy consumption, meeting the required service quality while reducing overall energy consumption through the optimal allocation of global resources.

5 Related Work

Hardware technology is employed in a large quantity of schemes on energy saving of current data center. As CPU's power consumption accounts for a larger proportion in the whole system, DVFS (Dynamic Voltage/Frequency Scaling) was introduced in [7], many studies [8, 9] are focused on the problem of high energy consumption of server CPU based on DVFS. The common approach is dynamic voltage scaling or requesting packet handling under low-load to reduce energy consumption. Some scholars [10, 11, 2] investigated the resource management strategies concerning the server cluster. Through dynamic reconfiguration (or contraction) of server cluster, the objective of saving energy by using fewer servers under low-load was achieved. However, these studies did not demonstrate the trade-off relationship between power consumption and performance from the perspective of the data center.

Besides, the placement problem itself is out of the scope of our work but the technique described in this paper can be helpful to any placement algorithm. Existing dynamic application placement proposals provide automation mechanisms by which resource allocations may be continuously adjusted to the changing workload. Previous work focuses on different goals, such as maximizing resource utilization [12] and their own service level goals [13, 5]. Our proposal can apply to and improve any of these placement solutions.

6 Conclusions and Future Work

Green data centers must achieve significantly lower power requirements and higher performance/watt ratio. In this paper, we identify new opportunities to improve the energy efficiency of data center, reducing the energy consumption, without negatively impacting the performance or user satisfaction. Our interest involves creating energy-aware framework EADC to contribute to building energy-efficient data centers. In the framework, we define the VE performance model using time-domain queue theory and reduce the DCMS utility using genetic algorithm. The obtained results show that EADC can achieve the optimal trade-off between performance and energy consumption.

It needs to further evolution of our EADC system both in real-world environment and prototype system. We are already working on the implementation of a real system that applies the techniques described in the paper. In the near future EADC system is to be introduced in the idea of autonomic computing so as to take better advantage of the resource availability.

Acknowledgements

This work is supported in part by the National High-Tech Research and Development Plan of China under grants No. 2009AA01A403, China Information Security Special (NDRC) fund for Disaster Backup and Recovery Standard

Architecture, and China Information Security Special fund (NDRC) for Disaster Backup and Recovery Product Industrialization.

References

- [1] Hartenstein, *Reconfigurable supercomputing: Hurdles and chances*, In Proceedings of the 2006 International Supercomputer Conference (ISC'06 June 28--30, Dresden, Germany) (invited article).
- [2] G. Chen, W. He, et al., *Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services*, In Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08), 2008, pp.337-350.
- [3] Haiping Qu, Xiuwen Wang, et al., *Research and Design of Deployment Framework for Blade-based Data Center*, In Proceedings of the 10th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2010)
- [4] Sergiu Nedeveschi , Sylvia Ratnasamy , Jitendra Padhye, *Hot data centers vs. cool peers*, Proceedings of the 2008 conference on Power aware computing and systems, p.8-8, December 07, 2008, San Diego, California.
- [5] XiaoYing Wang, ZhiHui Du, YiNong Chen, et al., *An Autonomic Provisioning Framework for Outsourcing Data Center Based on Virtual Appliances*, Cluster Computing, Volume 11, Issue 3, 229-245, September 2008.
- [6] <http://ita.ee.lbl.gov/>
- [7] M.Weiser, B.Welch, et al., *Scheduling for Reduced CPU Energy*, In OSDI (1994).
- [8] Kirk W. Cameron, Rong Ge, Xizhou Feng, *High-Performance, Power-Aware Distributed Computing for Scientific Applications*, IEEE Computer, Volume 38. No. 11, 2005, Pages 40-47.
- [9] Rong Ge, Xizhou Feng, and Kirk W. Cameron, *Performance-constrained, Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters*, 17th International ACM/IEEE Conference on High Performance Computing and Communications (SC 2005), November 2005. (Seattle, WA)
- [10] Y. Chen, A. Das, et al., *Managing Server Energy and Operational Costs in Hosting Centers*, SIGMETRICS'05, June, Canada, pp.303-314.
- [11] L. A. Barroso, U. Hölzle, *The Case for Energy-Proportional Computing*, IEEE Computer, Vol.40, No.12, Dec. 2007, pp.33-37.
- [12] A. Karve, T. Kimbrel, G. Pacifici, M. Spreitzer, M. Steinder, M. Sviridenko, A. Tantawi, *Dynamic placement for clustered web applications*, In WWW Conf., Scotland (2006).
- [13] M. Steinder, I. Whalley, D. Carrera, I. Gaweda and D.Chess, *Server virtualization in autonomic management of heterogeneous workloads*, In 10th IFIP/IEEE International Symposium on Integrated Management (IM 2007), May 2007.

SESSION

**COMMUNICATION SYSTEMS +
INTERCONNECTION NETWORKS**

Chair(s)

TBA

A Performance Metric for Message Forwarding Schemes of Massively Multiplayer Peer-to-Peer Based Networked Virtual Environments

James Dean Mathias¹ and Daniel Watson²

¹Dept. of Computer Science, Utah State University, Logan, UT 84322-4205, USA

²Dept. of Computer Science, Utah State University, Logan, UT 84322-4205, USA

Abstract—*This paper introduces a performance metric for use in characterizing message forwarding schemes of Massively Multiplayer Peer-to-Peer (P2P) based Networked Virtual Environments (NVE). Message forwarding, or routing, is the process by which peers send messages throughout an overlay. Different P2P NVE designs result in different message forwarding schemes; therefore, a metric is needed to evaluate the impact of a design on message forwarding performance. The metric presented in this paper was developed in the context of a project to determine the best Login Forwarding scheme for a Voronoi-based P2P overlay. The primary contribution of this paper is the performance metric, with a secondary contribution being the characterization of several Login Forwarding schemes.*

Keywords: Peer-to-Peer, Distributed Systems, Massive Multiplayer Online, Massive Virtual Environments, Networked Virtual Environments

1. Introduction

Communication between peers is a key concern in P2P-based Networked Virtual Environments (NVE). In such systems no peer knows about all other peers; therefore, a message forwarding scheme that overcomes this challenge is necessary. There are many different P2P NVE designs, with each impacting the performance of a messaging scheme differently. To date, the only reported evaluation technique of these schemes is the number of *hops* a message takes to arrive at its destination – an insufficient measure.

Client-server designs have a relatively simple communication scheme. A client sends a message, destined for another client, to the server. Because the server has a direct connection to every client, it sends the message directly to the destination client. All client-server systems share this same basic design, resulting in no differentiation in communication performance.

P2P systems differ significantly from client-server systems in the formation of their network overlay, resulting in differing messaging performance. In a client-server system, the number of connected clients has no impact on the number of hops between any client. P2P network overlays, on the other hand, change with every peer, which connects

or disconnects. Additionally, some P2P network overlays change structure as peers change position within the virtual environment (VE). P2P systems are far more complex in their communication structure than client-server systems, and therefore, demand a more sophisticated evaluation basis.

To further illustrate the issue, consider a P2P design and messaging scheme that results in a peer, or peers, being overwhelmed with message forwarding requests. For a content distribution network, this is a relatively minor inconvenience for the users. On the other hand, for an interactive Massively Multiplayer Online (MMO) system, overwhelming the bandwidth of a peer negatively impacts a user's experience. This may result in that user disconnecting from the network, with the problem moving to another peer and cascading as the problem persists. Using the number of hops as the only evaluation criteria, the problem remains hidden until too late.

Section 2 provides an overview of techniques others have used to evaluate message forwarding. The performance metric is detailed in Section 3. Section 4 describes the context in which the performance metric was originally developed. The experimental setup is presented in Section 5. The results from the simulation experiments are discussed in Section 6 and closing remarks in Section 7.

2. Related Work

In this section we review the message forwarding choices of representative P2P systems, along with the reported performance basis used in their evaluation. Put simply, performance evaluation of message forwarding schemes has not been properly addressed; therefore, little work exists.

Two projects under the name of Solipsis have been published [1], [2]. Both solutions rely upon a greedy message forwarding scheme, with neither paper presenting a basis for evaluation. Similarly, the VON framework [3] utilizes a greedy forwarding scheme. The authors do not individually evaluate this scheme; instead, any performance impact is aggregated into overall communication bandwidth performance.

The most common P2P messaging scheme employed by massive P2P NVE systems is Pastry [4]. Upon joining a

network, Pastry assigns a randomly selected 128-bit identifier to each peer. Based upon this identifier, other peers are able to use a distributed hash table (DHT) algorithm that allows peers to send messages between each other within $O(\log_2^b N)$ hops, where b is a configurable parameter, typically 4. Rowstron, et al., evaluate the performance of Pastry exclusively through the use of the number of hops as compared to the number of nodes.

The Peer Clustering prototype [5] uses a Pastry-based message forwarding scheme, with the authors reporting performance in terms of number of hops. Knutsson, et al., also used a Pastry-based scheme [6], again reporting performance in terms of the number of hops between network peers. Another scheme proposing to use Pastry is Mediator [7]. Because the paper is a proposal, there is no presentation of messaging performance.

Dickey, et al., present an event ordering technique using N-Trees [8]. Event ordering relies upon messaging between peers in order to resolve the ordering. The performance measure used to evaluate the cost of messaging in this scheme was number of peers in the network versus number of messages required.

3. Performance Metric

The performance of a messaging scheme is evaluated through the aggregation and summarization of data from messaging throughout the network, rather than for any single message. In other words, a messaging scheme is evaluated by sending many (thousands) messages throughout a network, with the results of those messages summarized into several performance measures. The metric is composed of the following measures:

- 1) Number of Hops Average/Median
- 2) Number of Hops Variance
- 3) Local Bandwidth Max
- 4) Local Bandwidth Average/Median
- 5) Local Bandwidth Variance
- 6) Global Bandwidth
- 7) Spatial Bandwidth Max
- 8) Spatial Bandwidth Variance

The number of hops a message takes is important because it is a proxy for how long a message takes to arrive at its destination. The average number of hops indicates the expected time to send a message, within the measured variance.

Local bandwidth indicates the bandwidth expectation at a peer. The Max value is the highest bandwidth usage by a single peer. The average, median, and variance are computed across all peers.

The peer with the maximum bandwidth demand is necessary in order to recognize the potential for demanding higher bandwidth at a peer than its expected available resources, potentially creating a highly negative user experience. The

average, median, and variance values indicate whether or not the messaging scheme is appropriate for the expected bandwidth resources available at a peer. The variance additionally indicates the fairness of the scheme. A scheme with a lower variance indicates the scheme requires similar resources from all peers. A higher variance indicates the scheme favors some peers over others, creating the potential for some peers to have an advantage because their networking demands are lower than others. The median is important because the data from messaging schemes isn't guaranteed to have a normal distribution. In these cases, the median bandwidth might be a better indicator of expected bandwidth demands.

Global bandwidth is the total number of hops taken for all messages recorded during the evaluation period.

Spatial Max and Variance are computed by subdividing the VE region into smaller square regions and aggregating results within each of these smaller regions. As a message is processed, the spatial location at which the processing occurred is recorded and added to the results for the subdivided region. The variance is computed over all the subdivided regions. For example, divide VE region into a grid of 100 x 100 smaller regions, creating 10,000 spatial regions in which data is collected.

The purpose of the Spatial Max and Variance is to reveal a spatial bias of the messaging scheme, if any. Whereas one scheme might not have a spatial bias, another may. A high spatial variance indicates the scheme exhibits a spatial bias. These values are computed by recording and binning messages based upon the VE location of the peer at the time they were logged. It isn't possible to specify the number of bins for any arbitrary VE; the expertise of the developer is still required to make a proper choice. While not specified in the metric, we additionally use a heatmap, which enables us to visually identify the nature of the spatial bias, if any. A messaging scheme that exhibits a spatial bias may lead to unintended social behaviors within the VE. As participants notice greater resource demands due to spatial locality, they will tend to avoid those locations, perhaps introducing additional performance problems with the scheme.

The number of messages, assuming messages are similarly sized, is a valid substitute for bandwidth.

4. Login Forwarding

The context of the performance metric presented in this paper is the evaluation of login forwarding techniques for our hybrid P2P NVE design, *Audrey*. Audrey is a Voronoi-based NVE, designed to host Massively Multiplayer virtual environments [9]. The framework includes a managed server, which is used for peer login and validation. As a peer joins the network overlay, it goes through several states before becoming an active participant. One of these states is known as *login forwarding*.

The login forwarding state involves a protocol through which a peer is forwarded to the correct overlay neighborhood, based upon its starting position in the VE. Login forwarding enables the joining peer to discover those neighbors with which it should be initially connected. The protocol begins with the joining peer contacting the managed server for forwarding. The server responds by sending contact details of an active peer to contact for further forwarding. The joining peer contacts this active peer to continue forwarding. This process repeats until the active peer closest to the joining peer's destination is discovered. Fundamentally, login forwarding is a messaging scheme.

Greedy Forwarding: A joining peer sends a forwarding request to another peer, the receiving peer. Upon receipt of a forwarding request, the receiving peer examines all its known neighbors, both Area of Interest (AOI) and enclosing, to find the one closest to the join destination, including the receiving peer itself. If the receiving peer is closest to the join destination, the joining peer is notified and the forwarding is complete. Otherwise, the contact information for the neighboring peer closest to the requested destination is sent to the joining peer. The joining peer continues the forwarding process by contacting the newly identified peer closest to its join destination.

Figure 1 illustrates a greedy forwarding sequence originating at the server peer and ending with a peer in the upper left corner of the virtual environment. The sequence begins with the server peer's Voronoi region highlighted, indicating it is the next receiving peer. The next step shows the enclosing (light grey) and AOI (dark grey) neighbors. From these neighbors, the one closest to the join location is selected as the next receiving peer; its Voronoi region is highlighted in the third step. The remaining steps illustrate the rest of the greedy forwarding sequence.

Because the purpose of Audrey is to enable massive peer participation, an efficient login forwarding protocol is needed. The naive approach to login forwarding is to use pure greedy forwarding, beginning at the server. As will be shown through our performance metric, this is also a poor choice. We identified several candidate techniques to improve upon pure greedy forwarding: three working set techniques and a grid based technique. Additionally, we included two techniques, FIFO and Random Selection, to help validate the effectiveness of the performance metric. The following list identifies these techniques, with the subsections that follow detailing each.

- 1) Best Case
- 2) Pure Greedy
- 3) First In, First Out (FIFO)
- 4) Random Selection
- 5) Working Set Random Replacement
- 6) Working Set Recent Replacement
- 7) Working Set Proportional
- 8) Grid Recent Replacement

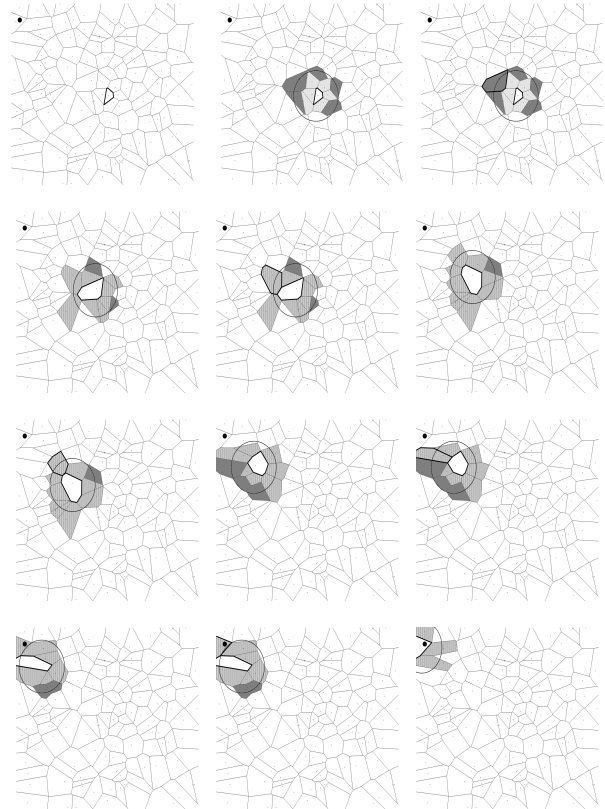


Fig. 1: Greedy Forwarding

For all techniques, once the initial peer is identified, greedy forwarding is employed to complete the join operation. The differentiating feature between each is the identification of the first peer to which the joining peer is handed off to begin greedy forwarding. Because the framework design is a P2P network, no single peer, including the server itself, has global knowledge of all active peer current locations. Therefore, the key to the best performance is to start the greedy forwarding process with the active peer as close to the destination as possible.

4.1 Best Case

Forwarding begins with the peer whose current location is closest to the destination of the joining peer. This provides the best possible selection. This is impossible in a real-world hybrid P2P deployment, because the server does not know the current location of all actively participating peers. However, under simulation conditions, it is possible to have global knowledge of the P2P overlay.

The concept is to provide a basis for evaluating how well any other variations approach the best case.

4.2 Pure Greedy

The design of Audrey specifies a bootstrapping peer, the server peer, located at the center of the virtual environment.

This peer has no virtual environment presence; its purpose is to provide the startup/fallback peer for the construction and maintenance of the P2P overlay. This variation specifies the server peer is selected, every time, as the node from which the greedy forwarding process begins; in other words, pure greedy forwarding.

The concept is that of a naive approach to handle login forwarding, without regard for efficiency or fairness.

4.3 First In, First Out

A first in, first out queue of active peers is maintained at the server. As a peer makes a forwarding request with the server, the peer at the front of the queue is selected as the starting peer for the greedy forwarding process. Once a peer completes forwarding, it is added to the end of the queue.

The concept is that of fairness of resource usage. Each peer must provide the same service it consumed for the next peer that joins the overlay. Fairness is emphasized over efficiency.

4.4 Random Selection

Forwarding starts by selecting a random peer from the set of known active peers.

The concept is that of fairness of resource usage, randomly distributing forwarding requests throughout all active peers. Fairness is emphasized over efficiency.

4.5 Working Set Random Replacement

The server maintains a fixed size set of active peers, the working set. The number of peers in the working set is relatively small, proportional to the total number of active peers. As a joining peer requests forwarding, the peer in the working set with its last known position closest to the joining peer's destination is chosen as the starting peer. The peer then selected to start the forwarding is removed from the working set and replaced by random selection from all known active peers. The number of peers in the working set is fixed throughout the lifetime of the server.

The concept is that of efficiency, with a secondary consideration with respect to fairness. Computational efficiency is considered by keeping a working set that is fixed in size and relatively smaller than all known active peers. Instead of testing every peer, a small number of peers are evaluated, ensuring a small, constant response time, even as the number of active peers increases. Efficiency with respect to global bandwidth is considered by choosing the peer with the last known position closest to the forwarding destination, the intention being to reduce the number of greedy forwarding requests required to join.

4.6 Working Set Recent Replacement

The technique has the same working set concept as described in Working Set Random Replacement, with a differing peer replacement scheme. The replacement peer is

selected by choosing the peer that has most recently become active. The number of peers in the working set is fixed throughout the lifetime of the server.

The concept in choosing the most recent active peer is that it is most likely closer to its starting location than any peer selected at random from all active peers. By choosing the most recent active peer, the replacement is in a similar location to the one replaced; this peer will also have the best, last known active location among all peers in the overlay. As this strategy is employed, the working set will contain peers with the most recent known active locations, distributed throughout the overlay.

4.7 Working Set Proportional

This is a variation on the Working Set Recent Replacement, differing in how the size of the working set is determined. The fixed sized working set is replaced by two parameters that control the size of a dynamically sized working set: 1) A minimum number of peers in the working set and 2) A maximum number of peers proportional to the number of active peers. The minimum specifies the smallest size the working set can ever be (given that number of active peers), while the maximum size changes in proportion to the number of active peers.

The concept is to grow and shrink the working set proportionally with the number of active peers, thereby dynamically changing the scope of the peers chosen from which to begin the greedy forwarding process.

4.8 Grid Recent Replacement

The server subdivides the virtual environment into a uniform grid of cells, identifying one peer for each of the cells from which the greedy login forwarding process begins. As a joining peer requests forwarding, the cell corresponding to the destination is computed and the peer within that cell is chosen as the starting peer. When a peer notifies the login server it has become active, the cell into which it belongs is computed and it becomes the forwarding peer for that cell until it is eventually replaced. Initially, the grid is populated with the server peer as the forwarding peer for each of the grid cells. As new peers become active, they replace the previous peer for their cell location. This creates turnover in the cells, helping to ensure the peer with the best last known position is represented within the grid. Therefore, the peer at all cell locations is the peer with the best last known position of any peer within that cell area.

Two parameters control this variation: 1) The starting size of the grid and 2) The threshold that causes the grid to increase in size. The grid is initialized to some size, for example, a 2x2 grid. As the number of active peers increases, the size of the grid also increases, thereby spreading out the distribution of peers from which forwarding can begin. For example, when the size of the grid is increased from 2x2 to 4x4, the peer at cell [0,0] from the 2x2 grid is replicated

Table 1: Simulation Parameters

AOI	AOI Range	Time Steps	New Peer
Fixed	1,000 units	5,000	1 @ 5 steps
Dynamic	10 peers	5,000	1 @ 5 steps
Working Set Parameters			
Technique	Working Set	Min Size	Ratio
Random	20	n/a	n/a
Recent	20	n/a	n/a
Proportional	n/a	10	0.05

into cells [0,0], [1,0], [0,1], [1,1] in the 4x4 grid. Over time, as new peers become active, they replace and create unique peers in the new grid.

The concept is to ensure a uniform distribution throughout the virtual environment of peers from which the greedy forwarding begins.

5. Experimental Setup

Simulations were performed to collect data in order to compare each of the techniques, using both fixed and dynamic AOI. Table 1 identifies the simulation parameters. The choice of 5,000 time steps was guided by previous work by Hu, et al. [3], where 3,000 time steps were used. In evaluating the usefulness of longer simulations for these techniques, some simulations were run with much longer time steps (over 30,000), which produced no difference in the results. Therefore, 5,000 was selected as having a proper balance of a long enough simulation to collect valid results, while providing short enough computation time to repeatedly run simulations. The choice to use 1,000 VE units and 10 max peers for the fixed and dynamic AOIs was guided by identifying parameters that show the differentiation between fixed and dynamic AOI. Finally, the choice to have 1 peer arrive every 5 time steps was made to ensure enough peers (1,000) joined the simulation to simulate a large number of active peers in the NVE.

At each time step, the simulation counts the number of active login forwarding messages contained within each peer's message queue; these data are used to compute the number of hops, local bandwidth, and global bandwidth measures. The VE is divided into a 100 x 100 grid of bins. At each time step, the number of login forwarding messages for all peers within that bin is recorded. These data are used to compute the spatial performance measures.

The number of neighbors tracked by each peer is a parameter that significantly affects the performance of login forwarding. This number is controlled by the AOI, of which two approaches are utilized, *fixed* and *dynamic*. Using fixed AOI, a peer tracks all neighbors within a fixed, circular region. Using dynamic AOI, a peer tracks a fixed number of peers regardless of their distance, with the circular AOI region defined by the distance to the furthest neighbor. For both approaches, all enclosing neighbors are tracked.

Table 2: Fixed AOI - Number of Hops

Technique	Average	Median	Std. Dev.
Best Case	8.59	8.00	2.27
Pure Greedy	24.24	25.00	8.81
FIFO	30.54	29.00	13.84
Random	28.91	29.00	13.93
WS Random	16.22	17.00	7.22
WS Recent	13.57	13.00	5.18
WS Proportional	13.29	13.00	5.14
Grid Recent	14.87	13.00	6.55

Table 3: Dynamic AOI - Number of Hops

Technique	Average	Median	Std. Dev.
Best Case	8.66	8.00	2.52
Pure Greedy	29.05	29.00	12.84
FIFO	36.23	22.00	18.56
Random	36.29	33.00	19.53
WS Random	18.39	17.00	9.59
WS Recent	16.58	13.00	8.25
WS Proportional	14.85	13.00	6.08
Grid Recent	18.36	17.00	10.68

The simulated virtual environment is a 10,000 by 10,000 unitless rectangular region. As a peer joins the simulation, its joining location within the VE is determined by the server using uniform random selection. The movement of each peer is also randomly determined. Initially, a random direction vector is selected, along with a randomly selected speed, and a randomly selected length of time for which the peer will move at that speed along the vector. Once the time period for that movement is complete, a new direction, speed, and length of time is selected; this is repeated for the lifetime of a peer.

Fixed AOI is not a scalable solution in a P2P environment due to the non-scalable number of messages required to maintain a P2P overlay. The number of messages required to support the overlay grows combinatorially with the number of neighbors within a peer's AOI. When using a fixed AOI, this number easily becomes a problem for network bandwidth utilization. The reason for showing fixed AOI results is to help understand the performance of dynamic AOI in comparison.

6. Results

Tables 2 through 6 report the performance metric measures from the simulations. Tables 2 and 3 report number of hops for the messages. Tables 4 and 5 report local and global utilization. Finally, Table 6 reports the spatial measures.

In addition to the simple measures of spatial maximum and variance, we have created heatmaps based upon the VE location of peers at the time the messages were processed. These data are visualized in Figures 2 and 3. The shape of the heat map corresponds to the rectangular region of the simulated virtual environment. Each data point represents the location of a peer at the time a message was processed. As the number of data points within an area accumulates, it is

Table 4: Peer Messages - Fixed AOI

Technique	Total	Average	Median	Std. Dev.	Max
Best Case	2,320	2.32	2	3.04	16
Pure Greedy	9,237	9.23	4	13.85	108
FIFO	13,144	13.13	8	14.17	79
Random	12,958	12.95	8	14.97	90
WS Random	6,815	6.81	5	7.19	45
WS Recent	5,506	5.5	4	5.15	32
WS Proportional	5,244	5.24	4	5.34	41
Grid Recent	6,055	6.05	4	6.8	48

Table 5: Peer Messages - Dynamic AOI

Technique	Total	Average	Median	Std. Dev.	Max
Base Case	2,274	2.27	2	2.97	18
Pure Greedy	12,268	12.26	6	16.58	140
FIFO	17,174	17.16	11	16.57	92
Random	16,765	16.75	12	16.5	110
WS Random	8,271	8.26	6	7.65	47
WS Recent	6,368	6.36	5	5.52	36
WS Proportional	6,259	6.25	5	5.8	37
Grid Recent	7,325	7.32	6	7.46	65

further darkened. Lighter regions represent areas of relatively few login forwarding messages, while darker regions represent areas with higher frequencies of messages. A visual inspection of the message distribution and density in the heat maps is confirmed by the data presented in the tables.

6.1 Results Discussion

Confidence in the validity of the metric is provided by the results of the Best Case technique in comparison to all others. In every performance measure, the Best Case is always better. The results of the performance measures, along with the heat maps, show differentiation among the techniques, illustrating the ability of the metric to differentiate performance among schemes.

Before running the simulations, we expected both FIFO and Random to be similar in performance, along with being the worst case scenarios; the metric validated this expectation. Peer starting positions are randomly selected throughout the VE; therefore, FIFO is essentially a random selection technique. The performance metric correctly captured this result.

Among the working set techniques, we expected the random replacement to be the least effective of the three; this was shown to be true. However, we expected proportional

Table 6: Spatial Messages

Technique	Fixed AOI		Dynamic AOI	
	Std. Dev.	Max	Std. Dev.	Max
Base Case	0.69	8	0.68	6
Pure Greedy	10.13	24	2.13	20
FIFO	2.02	94	2.21	66
Random	1.82	61	2.18	80
WS Random	1.15	8	1.23	17
WS Recent	1.00	13	1.13	16
WS Proportional	0.96	8	1.09	21
Grid Recent	1.12	27	1.23	28

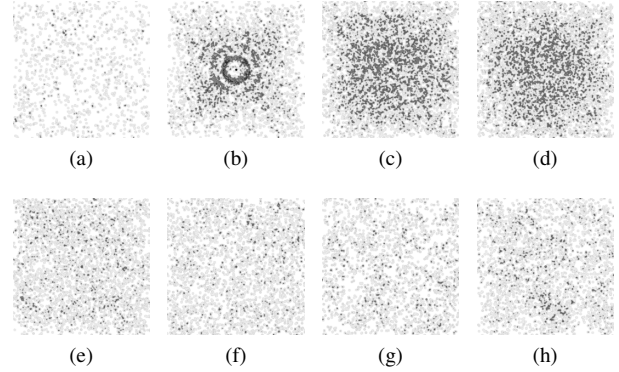


Fig. 2: Fixed AOI Messages. (a) Best Case (b) Pure Greedy (c) FIFO (d) Random (e) Working Set Random (f) Working Set Recent (g) Working Set Proportional (h) Grid Recent

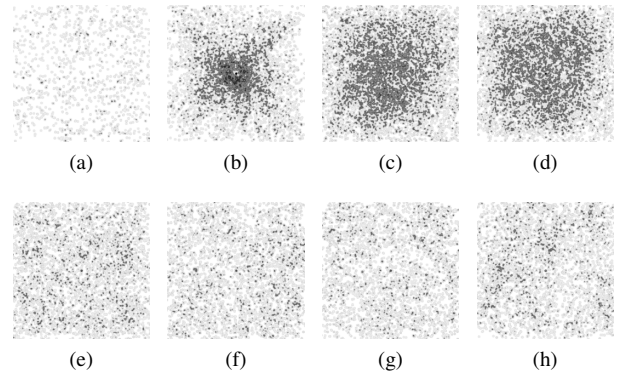


Fig. 3: Dynamic AOI Messages. (a) Best Case (b) Pure Greedy (c) FIFO (d) Random (e) Working Set Random (f) Working Set Recent (g) Working Set Proportional (h) Grid Recent

replacement to perform much better than recent replacement; the data do not show a statistically significant difference. An important differentiation feature of the proportional replacement technique is its computational complexity. As the number of active peers grows, the computational complexity grows linearly. Given that the fixed size recent replacement technique performs nearly as well with constant computational complexity, the choice between the two becomes obvious. The metric doesn't offer this insight; the expertise and knowledge of the developer is still required.

Before the simulations were performed, there was some debate regarding the performance of the grid replacement technique relative to the working set techniques. The metric shows its performance is better than the working set random replacement, but worse than the recent and proportional replacement techniques.

Note the high spatial variance under fixed AOI for the pure greedy technique and the circular pattern seen in Figure 2b.

The concentric circle radii approximate multiples of the AOI range from the server. The first circle lies along the outer boundary of the server's AOI, the second circle is two times that distance, and so on. This clearly illustrates the algorithm choosing the known neighbor closest to the destination of the joining peer, exhibiting a spatial bias the performance measure numbers don't readily demonstrate. In comparing Figures 2b and 3b, the behavior difference between the fixed and dynamic AOI is seen. For the dynamic AOI simulations, the number of neighbors was kept relatively low (10), which results primarily in choosing enclosing neighbors for handing off the login forwarding to the next peer. As all peers are constantly in motion, no particular fixed distance from one peer to its furthest neighbor exists, unlike fixed AOI where it is likely for one to have AOI neighbors near the fixed AOI distance.

7. Closing Remarks

This metric should be applied to individual message sub-systems and to all message sub-systems combined. The combined effects of messaging systems describes performance in general, but it is important to decompose the results by individual messaging schemes. Login forwarding, movement updates, broadcasts, and other systems must be individually characterized in order to correctly identify which systems are contributing to various performance measures.

The primary contribution of this paper is a performance metric that represents an important step forward in the characterization of P2P-based virtual environment messaging schemes. The performance measures offer a developer deeper insight into the side effects a scheme has locally among the peers, along with global effects such as spatial bias. Expertise on the part of the developer is still required to interpret the results, along with knowing what additional measures may be required.

The secondary contribution of this paper is the characterization, through the application of this performance metric, of several candidate login forwarding techniques. The results suggest the working set approach is the most promising direction in which to continue additional research. We anticipate further application of this metric as development continues on our hybrid P2P framework.

References

- [1] J. Keller and G. Simon, "Solipsis: A massively multi-participant virtual world," in *Parallel and Distributed Processing Techniques and Applications*. CSREA Press, 2003, pp. 262–268.
- [2] D. Frey, J. Royan, R. Piegay, A. Kermarrec, E. Anceaume, and F. Fessant, "Solipsis: A decentralized architecture for virtual environments," in *Workshop on Massively Multiuser Virtual Environments*, 2008.
- [3] S.-Y. Hu and G.-M. Liao, "Scalable peer-to-peer networked virtual environment," in *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA: ACM, 2004, pp. 129–133.
- [4] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. London, UK: Springer-Verlag, 2001, pp. 329–350.
- [5] A. Chen and R. R. Muntz, "Peer clustering: a hybrid approach to distributed virtual environments," in *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA: ACM, 2006, p. 11.
- [6] B. Knutsson, M. M. Games, H. Lu, W. Xu, and B. Hopkins, "Peer-to-peer support for massively multiplayer games," 2004.
- [7] L. Fan, H. Taylor, and P. Trinder, "Mediator: a design framework for p2p mmogs," in *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA: ACM, 2007, pp. 43–48.
- [8] C. GauthierDickey, V. Lo, and D. Zappala, "Using n-trees for scalable event ordering in peer-to-peer games," in *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video*. New York, NY, USA: ACM, 2005, pp. 87–92.
- [9] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Computing Surveys*, vol. 23, no. 3, pp. 345–405, 1991.

A New Property of Interconnection Networks

Yuan-Kang Shih¹, Jimmy J. M. Tan¹, and Lih-Hsing Hsu²

¹Department of Computer Science

National Chiao Tung University, Hsinchu, Taiwan 30010, R.O.C.

E-mail: {ykshih, jmtan}@cs.nctu.edu.tw.

²Department of Computer Science and Information Engineering

Providence University, Taichung, Taiwan 43301, R.O.C.

E-mail: lhhsu@cs.pu.edu.tw.

Abstract—A graph G is *pancyclic* if G includes cycles of all lengths and G is *edge-pancyclic* if each edge lies on cycles of all lengths. A bipartite graph is *edge-bipancyclic* if each edge lies on cycles of every even length from 4 to $|V(G)|$. Two cycles with the same length m , $C_1 = \langle u_1, u_2, \dots, u_m, u_1 \rangle$ and $C_2 = \langle v_1, v_2, \dots, v_m, v_1 \rangle$ passing through an edge (x, y) are independent with respect to the edge (x, y) if $u_1 = v_1 = x$, $u_m = v_m = y$ and $u_i \neq v_i$ for $2 \leq i \leq m - 1$. Cycles with equal length C_1, C_2, \dots, C_n passing through an edge (x, y) are mutually independent with respect to the edge (x, y) if each pair of them are independent with respect to the edge (x, y) . We propose a new concept called *mutually independent edge-bipancyclicity*. We say that a bipartite graph G is *k-mutually independent edge-bipancyclic* if for each edge $(x, y) \in E(G)$ and for each even length l , $4 \leq l \leq |V(G)|$, there are k cycles with the same length l passing through edge (x, y) , and these k cycles are mutually independent with respect to the edge (x, y) . In this paper, we prove that the hypercube Q_n is $(n - 1)$ -mutually independent edge-bipancyclic for $n \geq 4$.

Keywords: hypercube, bipancyclic, edge-bipancyclic, mutually independent

1. Introduction

For the graph definitions and notations we refer the reader to [1]. A graph is denoted by G with the vertex set $V(G)$ and the edge set $E(G)$. The simulation of one architecture by another is an important issue in interconnection networks. The problem of simulating one network by another is also called embedding problem. One particular problem of ring embedding deals with finding all the possible length of cycles in an interconnection network [2], [3], [4].

A path $P = \langle v_0, v_1, \dots, v_m \rangle$ is a sequence of adjacent vertices. We also write $P = \langle v_0, \dots, v_i, Q, v_j, \dots, v_m \rangle$ where Q is a path $\langle v_i, \dots, v_j \rangle$. A cycle $C = \langle v_0, v_1, \dots, v_m, v_0 \rangle$ is a sequence of adjacent vertices. The length of a path P is the number of edges in P . The length of a cycle C is the number of edges in C .

A path is a *hamiltonian path* if it contains all the vertices of G . A graph G is *hamiltonian connected* if there exists a hamiltonian path between any two different vertices of G . A graph $G = (B \cup W, E)$ is *bipartite* if $V(G)$ is the union of two disjoint sets B and W such that every edge joins B with W . It is easy to see that any bipartite graph with at least three vertices is not hamiltonian connected. A bipartite graph G is *hamiltonian laceable* if there exists a hamiltonian path joining any two vertices from different partite sets. A graph G is *pancyclic* [1] if G includes cycles of all lengths. A graph G is called *edge-pancyclic* if each edge lies on cycles of all lengths. If these cycles are restricted to even length, G is called a *bipancyclic graph*. A bipartite graph is *edge-bipancyclic* [5] if each edge lies on cycles of every even length from 4 to $|V(G)|$. A graph is *panconnected* if, for any two different vertices x and y , there exists a path of length l joining x and y , for every l , $d_G(x, y) \leq l \leq |V(G)| - 1$. The concept of panconnected graphs is proposed by Alavi and Williamson [6]. It is not hard to see that any bipartite graph with at least 3 vertices is not panconnected. Therefore, the concept of bipanconnected graphs is proposed. A bipartite graph is *bipanconnected* if, for any two different vertices x and y , there exists a path of length l joining x and y , for every l , $d_G(x, y) \leq l \leq |V(G)| - 1$ and $(l - d_G(x, y))$ being even. It is proved that the hypercube is bipanconnected [7].

We now introduce a relatively new concept. Two paths $P_1 = \langle u_1, u_2, \dots, u_m \rangle$ and $P_2 = \langle v_1, v_2, \dots, v_m \rangle$ from a to b are *independent* [8] if $u_1 = v_1 = a$, $u_m = v_m = b$, and $u_i \neq v_i$ for $2 \leq i \leq m - 1$. Paths with equal length P_1, P_2, \dots, P_n from a to b are *mutually independent* [8] if every two different paths are independent. Two paths P_1 and P_2 are *fully independent* [9] if $u_i \neq v_i$ for all $1 \leq i \leq m$. Paths with equal length P_1, P_2, \dots, P_n , are *mutually fully independent* if each pair of them are fully independent. Two cycles $C_1 = \langle u_1, u_2, \dots, u_m, u_1 \rangle$ and $C_2 = \langle v_1, v_2, \dots, v_m, v_1 \rangle$ passing through an edge (x, y) are *independent with respect to the edge (x, y)* , if $u_1 = v_1 = x$, $u_m = v_m = y$ and $u_i \neq v_i$ for $2 \leq i \leq m - 1$. Cycles with equal length C_1, C_2, \dots, C_n passing through an edge (x, y) are *mutually independent with respect to the edge (x, y)* if every two cycles are independent with respect

to the edge (x, y) .

An n -dimensional hypercube, denoted by Q_n , is a graph with 2^n vertices, and each vertex u can be distinctly labeled by an n -bit binary string, $u = u_{n-1}u_{n-2}\dots u_1u_0$. There is an edge between two vertices if and only if their binary labels differ in exactly one bit position. Let (u, v) be an edge in Q_n . If the binary labels of u and v differ in i th position, then the edge between them is said to be in i th dimension and the edge (u, v) is called an i th dimension edge. We use Q_n^0 to denote the subgraph of Q_n induced by $\{u \in V(Q_n) \mid u_i = 0\}$ and Q_n^1 the subgraph of Q_n induced by $\{u \in V(Q_n) \mid u_i = 1\}$. Q_n^0 and Q_n^1 are all isomorphic to Q_{n-1} . Q_n can be decomposed into Q_n^0 and Q_n^1 by dimension i , and Q_n^0 and Q_n^1 are $(n-1)$ -dimensional subcubes of Q_n induced by the vertices with the i th bit position being 0 and 1 respectively. For each vertex u in Q_n^i , $i = 0, 1$, there is exactly one vertex in Q_n^{i-1} , denoted by \bar{u} , such that (u, \bar{u}) is an edge in Q_n . Saad and Schultz [10] proved Q_n is edge-bipancyclic in the sense that each edge lies on cycles of every even length from 4 to 2^n . Li et al. [7] considered an injured n -dimensional hypercube Q_n where each edge lies on cycles of every even length from 4 to 2^n with $n-2$ edge faults. Tsai [11] proved that such injured hypercube Q_n contains a cycle of every even length from 4 to 2^n , even if it has up to $(2n-5)$ edge faults with some specified conditions. Sun et al. [12] proved that the n -dimensional hypercube Q_n contains $n-1$ mutually independent hamiltonian paths between any vertex pair $\{x, y\}$, where x and y belong to different partite sets and $n \geq 4$. Let $|F|$ be the number of the faulty edges. Hsieh and Weng [13] showed that when $1 \leq |F| \leq n-2$, there exists $n - |F| - 1$ mutually independent hamiltonian paths joining x to y in $Q_n - F$, where x and y belong to different partite sets.

We now introduce a new concept. We say that a bipartite graph G is n -mutually independent edge-bipancyclic if for each edge $(x, y) \in E(G)$, and for each even length l , $4 \leq l \leq |V(G)|$, there are n cycles with the same length l passing through edge (x, y) , and these n cycles are mutually independent with respect to the edge (x, y) . In this paper, we show that the hypercube has a stronger property of edge-bipancyclic property. We prove that an n -dimensional hypercube Q_n , for $n \geq 4$, is $(n-1)$ -mutually independent edge-bipancyclic in the sense that each edge of Q_n lies on $n-1$ mutually independent cycles of every even length from 4 to 2^n . Our result strengthens a previous result of Saad and Schultz [10]. Because each vertex of the hypercube Q_n has exactly n edges incident with it, we can expect at most $n-1$ mutually independent cycles passing through edge (x, y) . Therefore, the result " $n-1$ " is tight.

2. Preliminaries

In order to prove our claim, we need the following previous results.

Lemma 1: [14] The hypercube Q_n is hamiltonian laceable for every positive integer n .

Lemma 2: [7] The hypercube Q_n is bipanconnected for $n \geq 2$.

The hypercube Q_n is known to be a bipartite graph. Let (B, W) be the vertex bipartition of Q_n . Edges e_1, e_2, \dots, e_n in a graph G are called *independent edges* if these edges are pairwise disjoint.

Lemma 3: [12] Let $\{e_i \mid 1 \leq i \leq n-1\}$ be any $n-1$ independent edges of Q_n with $n \geq 2$ and $e_i = (b_i, w_i)$. Then there exist $n-1$ mutually fully independent hamiltonian paths P_1^l, \dots, P_{n-1}^l of Q_n such that P_i^l joins from b_i to w_i .

Theorem 1: [12] Let x and y be two vertices from different partite sets of Q_n , for $n \geq 4$. Then there exist $n-1$ mutually independent hamiltonian paths joining x to y .

Theorem 2: [15] Let F_v be a set of faulty vertices in Q_n . There exists a path of every odd length from 3 to $2^n - 2|F_v| - 1$ joining any two adjacent fault-free vertices in $Q_n - F_v$ even if $|F_v| \leq n-2$, where $n \geq 3$.

Lemma 4: [12] $Q_n - \{x, y\}$ is hamiltonian laceable, if x and y are any two vertices from different partite sets of Q_n with $n \geq 4$.

3. Main Results

To prove our main result, we need the following lemmas, Lemma 5 to 7.

Lemma 5: Let x and y be two vertices from different partite sets of Q_n with $n \geq 4$. There exists a path of every odd length from 1 to $2^n - 3$ joining any two adjacent fault-free vertices in $Q_n - \{x, y\}$.

Proof: Let u, v be two adjacent fault-free vertices in $Q_n - \{x, y\}$. Because u and v are adjacent fault-free vertices, there exists a path of length 1 joining from u to v in $Q_n - \{x, y\}$. According to Theorem 2, there exists a path of every odd length from 3 to $2^n - 2|2| - 1 (= 2^n - 5)$ joining u to v in $Q_n - \{x, y\}$. Then by Lemma 4, there exists a path of length $2^n - 3$ joining u to v in $Q_n - \{x, y\}$. Therefore, the lemma holds. ■

Lemma 6: Let e_1 and e_2 be two independent edges of Q_3 , $e_i = (b_i, w_i)$ for $i = 1, 2$. Then Q_3 contains 2 mutually fully independent paths P_1^l and P_2^l with any odd length $l \leq 2^3$ such that P_i^l joins from b_i to w_i , $i = 1, 2$.

Proof: By the edge-symmetric property of hypercubes, we may assume that $e_1 = (000, 100)$. We then consider the following three cases, $e_2 =$

Table 1: The proof of Lemma 6

$e_1 = (000, 100)$ $e_2 = (011, 001)$	$l = 1$	$P_1^1 = \langle 000, 100 \rangle$ $P_2^1 = \langle 011, 001 \rangle$
	$l = 3$	$P_1^3 = \langle 000, 010, 110, 100 \rangle$ $P_2^3 = \langle 011, 111, 101, 001 \rangle$
		$l = 5$
	$l = 7$	
$e_1 = (000, 100)$ $e_2 = (011, 111)$	$l = 1$	$P_1^1 = \langle 000, 100 \rangle$ $P_2^1 = \langle 011, 111 \rangle$
	$l = 3$	$P_1^3 = \langle 000, 010, 110, 100 \rangle$ $P_2^3 = \langle 011, 001, 101, 111 \rangle$
		$l = 5$
	$l = 7$	
$e_1 = (000, 100)$ $e_2 = (101, 001)$	$l = 1$	$P_1^1 = \langle 000, 100 \rangle$ $P_2^1 = \langle 101, 001 \rangle$
	$l = 3$	$P_1^3 = \langle 000, 010, 110, 100 \rangle$ $P_2^3 = \langle 101, 111, 011, 001 \rangle$
		$l = 5$
	$l = 7$	

$\{(011, 001), (011, 111), (101, 001)\}$. For each of these cases, we construct the required paths, see Table 1. ■

Lemma 7: Let $\{e_i \mid 1 \leq i \leq n-1\}$ be $n-1$ independent edges of Q_n with $n \geq 2$, $e_i = (b_i, w_i)$, $i = 1$ to $n-1$. Then there exist $n-1$ mutually fully independent paths P_1^l, \dots, P_{n-1}^l of Q_n with any odd length $l \leq 2^n - 1$ such that P_i^l joins from b_i to w_i , $i = 1$ to $n-1$.

Proof: It is clear that the result holds for Q_2 . We prove the statement by induction on n . By Lemma 6, the statement holds for $n = 3$. Suppose that the result holds for Q_{n-1} , for some $n \geq 4$. The hypercube Q_n has n dimensions, and there are only $n-1$ independent edges, so there is at least one dimension which does not contain any one of these $n-1$ independent edges. We can choose one of these dimensions to separate Q_n into two $(n-1)$ -dimensional subcubes Q_n^0 and Q_n^1 . We then prove the result by considering the following three cases.

Case 1. For odd length l and $1 \leq l \leq 2^{n-1} - 1$.

Case 1.1. Suppose that there are k independent edges in Q_n^0 with $1 \leq k \leq n-2$ and there are $n-k-1$ independent edges in Q_n^1 . By induction hypothesis, the case is obvious.

Case 1.2. Without loss of generality, suppose that all the $n-1$ independent edges are in Q_n^0 . By induction hypothesis, there exist $n-2$ mutually fully independent paths P_1^l, \dots, P_{n-2}^l of Q_n^0 with any odd length $l \leq 2^{n-1} - 1$ such that P_i^l joins from b_i to w_i for $1 \leq i \leq n-2$. By Lemma 2, there is a path R_{n-1}^m of Q_n^1 with odd length $1 \leq m \leq 2^{n-1} - 3$ joining \bar{b}_{n-1} to \bar{w}_{n-1} . Let $P_{n-1}^l = \langle b_{n-1}, \bar{b}_{n-1}, R_{n-1}^m, \bar{w}_{n-1}, w_{n-1} \rangle$, then $3 \leq l \leq 2^{n-1} - 1$. Note that, b_{n-1} and w_{n-1} are adjacent vertices, so we obtain paths P_{n-1}^l for all odd lengths l , $1 \leq l \leq 2^{n-1} - 1$. Therefore, there are $n-1$ mutually fully independent paths P_1^l, \dots, P_{n-1}^l of Q_n with any odd length $l \leq 2^n - 1$ such

that P_i^l joins from b_i to w_i , $i = 1$ to $n-1$.

Case 2. For odd length l and $2^{n-1} + 1 \leq l \leq 2^n - 3$.

Case 2.1. Suppose that there are k independent edges in Q_n^0 with $1 \leq k \leq n-2$ and there are $n-k-1$ independent edges in Q_n^1 . By induction hypothesis, there exist k mutually fully independent paths R_1, \dots, R_k of Q_n^0 with length $2^{n-1} - 1$ such that R_i joins from b_i to w_i for $1 \leq i \leq k$. We let $R_i = \langle b_i, u_i, v_i, Z_i, w_i \rangle$ for $1 \leq i \leq k$. According to induction hypothesis, there exist k mutually fully independent paths $T_1^{l'}, \dots, T_k^{l'}$ of Q_n^1 with any odd length $l' \leq 2^{n-1} - 3$ such that $T_i^{l'}$ joins from \bar{u}_i to \bar{v}_i for $1 \leq i \leq k$. Therefore, $P_i^l = \langle b_i, u_i, \bar{u}_i, T_i^{l'}, \bar{v}_i, v_i, Z_i, w_i \rangle$ with $2^{n-1} + 1 \leq l \leq 2^n - 3$ for $1 \leq i \leq k$. Again by induction hypothesis, there exist $n-k-1$ mutually fully independent paths R_{k+1}, \dots, R_{n-1} of Q_n^1 with length $2^{n-1} - 1$ such that R_i joins from b_i to w_i for $k+1 \leq i \leq n-1$. We let $R_i = \langle b_i, u_i, v_i, Z_i, w_i \rangle$ for $k+1 \leq i \leq n-1$. By induction hypothesis, there exist $n-k-1$ mutually fully independent paths $T_{k+1}^{l'}, \dots, T_{n-1}^{l'}$ of Q_n^0 with any odd length $l' \leq 2^{n-1} - 3$ such that $T_i^{l'}$ joins from \bar{u}_i to \bar{v}_i for $k+1 \leq i \leq n-1$. Therefore, $P_i^l = \langle b_i, u_i, \bar{u}_i, T_i^{l'}, \bar{v}_i, v_i, Z_i, w_i \rangle$ with $2^{n-1} + 1 \leq l \leq 2^n - 3$ for $k+1 \leq i \leq n-1$. Hence, there are $n-1$ mutually fully independent paths P_1^l, \dots, P_{n-1}^l of Q_n with any odd length $2^{n-1} + 1 \leq l \leq 2^n - 3$ such that P_i^l joins from b_i to w_i .

Case 2.2. Without loss of generality, suppose that all the $n-1$ independent edges are in Q_n^0 . By induction hypothesis, there exist $n-2$ mutually fully independent paths R_1, \dots, R_{n-2} of Q_n^0 with length $2^{n-1} - 1$ such that R_i joins from b_i to w_i for $1 \leq i \leq n-2$. We let $R_i = \langle b_i, Z_i, u_i, v_i, z_i, w_i \rangle$. Again by induction hypothesis, there exist $n-2$ mutually fully independent paths $T_1^{l'}, \dots, T_{n-2}^{l'}$ of Q_n^1 with any odd length $l' \leq 2^{n-1} - 3$ such that $T_i^{l'}$ joins from \bar{u}_i to \bar{v}_i for $1 \leq i \leq n-2$. Therefore, $P_i^l = \langle b_i, Z_i, u_i, \bar{u}_i, T_i^{l'}, \bar{v}_i, v_i, z_i, w_i \rangle$ with any odd length $2^{n-1} + 1 \leq l \leq 2^n - 3$ for $1 \leq i \leq n-2$. By Lemma 2, there exists a path R_{n-1} of Q_n^1 with length $2^{n-1} - 3$ joining \bar{b}_{n-1} to \bar{w}_{n-1} . We let $R_{n-1} = \langle \bar{b}_{n-1}, Z_{n-1}, u_{n-1}, v_{n-1}, \bar{w}_{n-1} \rangle$. By Lemma 5, there exists a path $T_{n-1}^{l'}$ with every odd length $1 \leq l' \leq 2^{n-1} - 3$ joining \bar{u}_{n-1} to \bar{v}_{n-1} in $Q_n^0 - \{b_{n-1}, w_{n-1}\}$. Therefore, $P_{n-1}^l = \langle b_{n-1}, \bar{b}_{n-1}, Z_{n-1}, u_{n-1}, \bar{u}_{n-1}, T_{n-1}^{l'}, \bar{v}_{n-1}, v_{n-1}, \bar{w}_{n-1}, w_{n-1} \rangle$ with $2^{n-1} + 1 \leq l \leq 2^n - 3$. So, there are $n-1$ mutually fully independent paths P_1^l, \dots, P_{n-1}^l of Q_n with any odd length $2^{n-1} + 1 \leq l \leq 2^n - 3$ such that P_i^l joins from b_i to w_i .

Case 3. For odd length l and $l = 2^n - 1$. This case is proved by Lemma 3.

By Case 1 Case 2 and Case 3, the proof is complete. ■

We now prove our main result by induction.

Lemma 8: The hypercube Q_4 is 3-mutually independent edge-bipancyclic.

Proof: By the edge-symmetric property of hypercubes, we may assume that the edge is (0000, 0001). We construct the required cycles in Table 2. ■

Table 2: The proof of Lemma 8

4 - cycle	(0000, 0100, 0101, 0001, 0000) (0000, 0010, 0011, 0001, 0000) (0000, 1000, 1001, 0001, 0000)
6 - cycle	(0000, 0100, 0110, 0111, 0101, 0001, 0000) (0000, 0010, 1010, 1011, 0011, 0001, 0000) (0000, 1000, 1100, 1101, 1001, 0001, 0000)
8 - cycle	(0000, 0100, 0110, 0010, 0011, 0111, 0101, 0001, 0000) (0000, 0010, 1010, 1110, 1111, 1011, 1001, 0001, 0000) (0000, 1000, 1001, 1011, 1010, 0010, 0011, 0001, 0000)
10 - cycle	(0000, 0100, 0110, 0010, 1010, 1011, 0011, 0111, 0101, 0001, 0000) (0000, 0010, 0011, 0111, 0110, 0100, 0101, 1101, 1001, 0001, 0000) (0000, 1000, 1010, 1110, 1100, 1101, 1111, 1011, 0011, 0001, 0000)
12 - cycle	(0000, 0100, 0110, 0010, 1010, 1110, 1111, 1011, 0011, 0111, 0101, 0001, 0000) (0000, 0010, 0011, 0111, 0110, 0100, 0101, 1101, 1100, 1000, 1001, 0001, 0000) (0000, 1000, 1100, 1101, 1111, 0111, 0110, 1110, 1010, 1011, 0011, 0001, 0000)
14 - cycle	(0000, 0100, 0101, 0111, 0011, 1011, 1010, 1000, 1100, 1101, 1001, 1011, 0011, 0001, 0000) (0000, 0010, 0110, 0100, 0101, 0111, 0011, 1011, 1010, 1000, 1100, 1101, 1001, 0001, 0000) (0000, 1000, 1010, 1011, 1001, 1101, 1100, 1110, 1111, 0111, 0110, 0100, 0101, 0001, 0000)
16 - cycle	(0000, 0100, 0101, 0111, 0110, 0010, 1010, 1000, 1100, 1110, 1111, 1011, 1001, 1011, 0011, 0001, 0000) (0000, 0010, 0110, 0100, 0101, 0111, 0011, 1011, 1010, 1000, 1100, 1110, 1111, 1101, 1001, 0001, 0000) (0000, 1000, 1010, 1011, 1001, 1101, 1100, 1110, 1111, 0111, 0011, 0010, 0110, 0100, 0101, 0001, 0000)

Theorem 3: The hypercube Q_n is $(n-1)$ -mutually independent edge-bipancyclic for $n \geq 4$.

Proof: Let (u, v) be an edge in Q_n , $n \geq 4$. We prove the statement by induction on n . By Lemma 8, the statement holds for $n = 4$. Suppose that the result holds for Q_{n-1} , $n \geq 5$. We may choose a dimension to divide the hypercube Q_n into two subcubes Q_n^0 and Q_n^1 so that the edge (u, v) is in Q_n^0 . According to the length l of the cycles, we divide the proof into the following three cases. In each case, the length l is assumed to be an even number. We shall find $n-1$ mutually independent cycles with length l passing through edge (u, v) .

Case 1. For even length l and $4 \leq l \leq 2^{n-1}$.

By induction hypothesis, there exist $n-2$ mutually independent cycles with respect to the edge (u, v) , C_1^k, \dots, C_{n-2}^l with any even length $4 \leq l \leq 2^{n-1}$ in Q_n^0 . By Lemma 2, there is a path P^k of Q_n^1 with any odd length $1 \leq k \leq 2^{n-1} - 3$ joining \bar{u} to \bar{v} . Then we have $C_{n-1}^l = \langle u, \bar{u}, P^k, \bar{v}, v, u \rangle$ with any even length $4 \leq l \leq 2^{n-1}$. Therefore, there exist $n-1$ mutually independent cycles with respect to the edge (u, v) , C_1^l, \dots, C_{n-1}^l with every even length $4 \leq l \leq 2^{n-1}$.

Case 2. For even length l and $2^{n-1} + 2 \leq l \leq 2^n - 2$.

By induction hypothesis, there exist $n-2$ mutually independent cycles with respect to the edge (u, v) , R_1, \dots, R_{n-2} with length 2^{n-1} of Q_n^0 . We let $R_i = \langle u, x_i, y_i, z_i, \dots, v, u \rangle$ for $1 \leq i \leq n-2$. By Lemma 7, for any given odd length $k \leq 2^{n-1} - 3$ there exist $n-2$ mutually fully independent paths P_1^k, \dots, P_{n-2}^k all with the same length k , such that P_i^k joins from \bar{y}_i to \bar{z}_i for $1 \leq i \leq n-2$. We

let $C_i^l = \langle u_i, x_i, y_i, \bar{y}_i, P_i^k, \bar{z}_i, z_i, R_i, v_i, u_i \rangle$. Then C_i^l , $i = 1$ to $n-2$ are with any even length l , where $2^{n-1} + 2 \leq l \leq 2^n - 2$. By Lemma 1, there exists a hamiltonian path P' of Q_n^1 joining \bar{u} to \bar{v} . Let $P' = \langle \bar{u}, y_{n-1}, z_{n-1}, T, \bar{v} \rangle$. By Lemma 5, there exists a path $U^{k'}$ with every odd length $1 \leq k' \leq 2^{n-1} - 3$ joining \bar{y}_{n-1} to \bar{z}_{n-1} in $Q_n^0 - \{u, v\}$. We let $C_{n-1}^l = \langle u, \bar{u}, y_{n-1}, \bar{y}_{n-1}, U^{k'}, \bar{z}_{n-1}, z_{n-1}, T, \bar{v}, v \rangle$ with any even length l , $2^{n-1} + 2 \leq l \leq 2^n - 2$. Hence, there exist $n-1$ mutually independent cycles with respect to edge (u, v) , C_1^l, \dots, C_{n-1}^l with any even length $4 \leq l \leq 2^{n-1}$.

Case 3. For even length l and $l = 2^n$. This case is proved by Theorem 1.

By Case 1, Case 2 and Case 3, we complete the proof. ■

4. Conclusion

In [1], the author introduced a popular property called the *pancyclicity*. A stronger property is *edge-bipancyclicity* which was proposed by Mitchem and Schmeichel in [5]. Another interesting property is the *mutually independent paths*. Sun et al. [12] proved that the n -dimensional hypercube graph contains $n-1$ mutually independent hamiltonian paths between any vertex pair $\{x, y\}$, where x and y belong to different partite sets and $n \geq 4$. In this paper, we combine the two properties, edge-bipancyclicity and mutually independent paths, into a new stronger property called *mutually independent edge-bipancyclic property*, and show that the hypercube Q_n is $(n-1)$ -mutually independent edge-pancyclic for $n \geq 4$. Our result also strengthens a previous result of Saad and Schultz [10], in the sense that the hypercube Q_n is not only edge-bipancyclic but also mutually independent edge-pancyclic.

Acknowledgements

This research was partially supported by the National Science Council of the Republic of China under contract NSC 99-2221-E-009-084-MY3 and the Aiming for the Top University and Elite Research Center Development Plan.

References

- [1] U. S. R. Murty, *Graph Theory with Applications*, London: Macmillan Press, 1976.
- [2] K. Day and A. Tripathi, "Embedding of Cycles in Arrangement Graph," *IEEE Transactions on Computer*, vol. 42, pp. 1002-1006, 1993.
- [3] A. Germa, M. C. Heydemann, and D. Sotteau, "Cycles in the Cubeconnected Cycles Graph," *Discrete Applied Mathematics*, vol. 83, pp. 135-155, 1998.
- [4] S. C. Hwang and G. H. Chen, "Cycle in Butterfly Graphs," *Networks*, vol. 35, pp. 161-171, 2000.
- [5] J. Mitchem and E. Schmeichel, "Pancyclic and Bipancyclic Graphs-a Survey," in *Proceeding. First Colorado Symposium on Graphs and Applications*, 1985, pp. 271-278.
- [6] Y. Alavi and J. E. Williamson, "Panconnected Graph," *Studia Scientiarum Mathematicarum Hungarica*, vol. 10, pp. 19-22, 1975.
- [7] T.-K. Li, C.-H. Tsai, J. J. M. Tan, and L.-H. Hsu, "Bipanconnectivity and Edge-fault-tolerant Bipancyclicity of Hypercubes," *Information Processing Letters*, vol. 87, pp. 107-110, 2003.

- [8] C.-K. Lin, H.-M. Huang, L.-H. Hsu, and S. Bau, "Mutually Independent Hamiltonian Paths in Star Networks," *Networks*, vol. 46, pp. 110–117, 2005.
- [9] S.-Y. Hsieh and P.-Y. Yu, "Fault-free Mutually Independent Hamiltonian Cycles in Hypercubes with Faulty Edges," *Journal of Combinatorial Optimization*, vol. 13, pp. 153–162, 2007.
- [10] Y. Saad and M. H. Schultz, "Topological Properties of Hypercubes," *IEEE Transactions on Computers*, vol. 37, pp. 867–872, 1988.
- [11] C. H. Tsai, "Linear array and ring embeddings in conditional faulty hypercubes," *Theoretical Computer Science*, vol. 314, pp. 431–443, 2004.
- [12] C.-M. Sun, C.-K. Lin, H.-M. Huang, and L.-H. Hsu, "Mutually Independent Hamiltonian Paths and Cycles in Hypercubes," *Journal of Interconnection Networks*, vol. 7, pp. 235–255, 2006.
- [13] S.-Y. Hsieh and Y.-F. Weng, "Fault-Tolerant Embedding of Pairwise Independent Hamiltonian Paths on a Faulty Hypercube with Edge Faults," *Theory of Computing Systems*, vol. 45, pp. 407–425, 2009.
- [14] G. Simmons, "Almost all n -dimensional Rectangular Lattices are Hamilton Laceable," *Congressus Numerantium*, vol. 21, pp. 103–108, 1978.
- [15] S.-Y. Hsieh and T.-H. Shen, "Edge-bipancyclicity of a Hypercube with Faulty Vertices and Edges," *Discrete Applied Mathematics*, vol. 156, pp. 1802–1808, 2008.

Audrey: The Model and Implementation of A Hybrid P2P Framework for Massive Virtual Environments

James Dean Mathias¹ and Daniel Watson²

¹Dept. of Computer Science, Utah State University, Logan, UT 84322-4205, USA

²Dept. of Computer Science, Utah State University, Logan, UT 84322-4205, USA

Abstract—*Massively Multiplayer Online environments continue to grow in popularity, with current technical designs based upon a well proven client-server model. This approach has some inherent limitations, high costs to provision server resources for peak demands and restriction of the maximum number of concurrent participants within a virtual environment. Incorporating Peer-to-Peer (P2P) techniques provides developers the opportunity to significantly reduce costs, while also breaking through the barrier of the number of concurrent participants within a single virtual environment. We propose a hybrid P2P model incorporating a managed server along with a Voronoi-based P2P overlay for the development of massive virtual environments. This paper explains the current limitations of both client-server and pure P2P systems and presents the design and implementation of our hybrid P2P system, which resolves some of these limitations.*

Keywords: Peer-to-Peer, Distributed Systems, Massive Multiplayer Online, Massive Virtual Environments, Networked Virtual Environments

1. Introduction

Massively Multiplayer Online (MMO) environments have become a significant component of the computer game industry, including titles such as Ultima Online, Everquest, Eve Online, and World of Warcraft. These represent a few names among a growing and popular landscape of MMOs. World of Warcraft dominates, with Blizzard boasting an active subscriber base of 12 million [1], while other popular MMOs have active subscriber bases in the range of several hundred thousand. MMOs represent a unique opportunity for the entertainment industry; namely, the ability to create persistent virtual worlds with participation counted in the thousands. This is in contrast to standard multi-player games that have no persistence and allow, at best, a few dozen participants in the same virtual environment (VE). While MMOs are tremendously popular, current commercial deployments are constrained by various technical limitations, affecting both the VE design and the cost to deliver and maintain.

The technical design of MMOs like World of Warcraft (WOW) is a client-server architecture. The VE simulation is performed at a server, or more specifically, at a server farm.

Every participant in the VE connects through a computer, with their computer acting as a client. The *client* renders the VE from the perspective of the participant, collects input, sends this input to the server, monitors the server for updates, and re-renders the VE based upon updated state information. The server continuously receives inputs from possibly many thousands of connected clients and uses these inputs to update the VE simulation. The *server* is the authoritative source for the current state of the VE simulation, disseminating updates to any connected clients.

At the Austin Game Developer Conference in September of 2009, Blizzard discussed the backend requirements used to support World of Warcraft [2]. They report 13,250 server blades running 75,000 CPU cores, spread across an unstated number of server farms. Estimating 200 CPU cores per server farm, this is on the order of 375 server farms, or server instances, as multiple server farms may be located in the same physical location. Using this number and working from their reported subscriber base, we can estimate a maximum allowable number of registered users per server instance, to be on the order of 50,000, with each instance allowing perhaps up to 10,000 active participants. Another popular MMO, Eve Online [3], claims to have the largest number of simultaneous users on a single server (farm) instance of just over 56,000 users. These numbers indicate that there is a high cost to purchase and maintain these computing resources, along with a limited number of concurrent users within a single VE. There is a high cost to purchase these computers, physically host them around the world, and pay for network bandwidth, power, cooling, human technical support, and other costs simply to have this level of managed physical resources. Of course, a company with a paying subscriber base of 12 million can afford these physical resources, but even large companies search for reductions in their expenses in order to maximize their profit. Similarly, smaller companies aim to reduce the upfront financial barrier.

Fundamentally, a client-server technical design results in a high-cost risk for any organization, in addition to representing a barrier to entry for smaller organizations. If an organization underestimates or overestimates demand, an over-allocation or under-allocation of resources results. In the case of underestimating demand, customers are frustrated because of poor performance, or an inability to even access

the MMO. In the case of overestimating demand, money is unnecessarily wasted in providing physical hosting resources, and very likely resulting in a sparsely-populated MMO. Both of these problems may lead to failure, due to high maintenance costs and low rates of participation, i.e. low revenue stream.

Another limitation imposed by the client-server architecture is the design of the VEs. Current server architectures limit the number of simultaneous users on any particular server instance. This technical limitation leads to a VE design requirement: The VE is designed to accommodate a tiny fraction of the total subscriber base, leading to the deployment of multiple VE instances in order to accommodate the total subscriber base. Removing this technical limitation frees designers to contemplate new VEs, enabling hundreds of thousands, potentially millions, of simultaneous users.

One approach to alleviating the technical limitations imposed by a client-server design is to use peer-to-peer (P2P) design elements. P2P networks enable an application to distribute computing tasks among the participating clients. By requiring each connected client to provide more computing resources than it consumes, the potential for a scalable computing environment exists. Traditionally, P2P networks have been used for content distribution. That is not their only use, they can be utilized for computational tasks, with those tasks originating and computed without centralized, global coordination.

This paper presents *Audrey*, a hybrid P2P model and operational framework for the simulation of massive networked virtual environments (NVE). An important distinguishing aspect of *Audrey*, from other P2P models, is that it is an operational, distributed framework that provides a scalable and secure computational overlay in which peers can participate. This is a critical step to move forward in massive P2P-NVE research and development, transitioning from simulations to operational systems.

The paper begins with a review of research with respect to P2P-based massive networked virtual environment models described in Section 2. Section 3 describes the design and hybrid model of our approach. Section 4 discusses the implementation of our model as an operational system. Section 5 details the different runtime environments used for the development and execution of our framework. Closing remarks are offered in Section 6.

2. Related Work

The field of P2P-based NVEs is an area of active research; however, with no clear winning strategy and there are no current commercial deployments. P2P NVE systems borrow from many fields of computing, including peer-to-peer systems, distributed computing, simulation, security, networking, and databases. It is the application of these fields in a unique combination that makes P2P NVE research a

distinct activity. The work reviewed in this section is related specifically to P2P-based NVEs.

An example of a P2P NVE design is the Mediator framework [4]. Mediator uses a structured P2P overlay, formed through distributed hashing, along with application level multicasting and direct P2P communication. The VE is decomposed into a number of rectangular zones based upon load management needs. These zones incorporate one or more super-peers, or Mediators, each of which may take on different roles. They then have a responsibility to maintain the overlay, VE simulation, and communication activities.

Solipsis [5] is a proposal for the design of a P2P-based NVE, outlining an approach for a globally-connected network and a login and teleportation protocol for finding the peer closest to a desired destination. The bulk of their study is concerned with the construction and proof of the overlay connectivity, with a short section describing a greedy forwarding mechanism for peer login and teleportation. In 2008, Frey, et al., presented a new architecture, under the name of Solipsis [6], which was a pure P2P approach, based upon an n-dimensional Voronoi overlay. Computationally intensive tasks, such as physics and collision detection, are distributed throughout the overlay instead of a central server. This project is unique in that it is an actual working framework; the architecture, unfortunately, does not consider the primary issue of security.

Another approach to deploying NVEs over a P2P network is to embrace the nature of P2P, as opposed to emulating a client-server model. Hughes, et al., [7] work from the assumption that P2P networks are unpredictable; in particular, they focus on P2P networks without a central authority. Their suggestion is to embrace this in the design of the game world rather than force predictability over the network. Their VE design is intimately tied to the underlying peers participating, with each peer hosting a room. As a peer is active, the room is available; when a peer fails to respond, that room is not accessible. Ultimately, the game is about exploration of the P2P environment instead of being a goal-directed game.

Knutsson, et al., present the design and results from their P2P research experiment, SimMud [8]. The SimMud architecture is built upon the Pastry [9] foundation, utilizing the Scribe [10] application level multicast infrastructure. The authors assert that their results show it is feasible to create a P2P-based massive NVE appropriate for the type of game designs typical MMOs employ.

The Voronoi Based Adaptive Scalable (VAST) project is an ongoing P2P NVE research project that has already produced many results [11], [12], [13]. Their work uses a Voronoi diagram to construct and maintain the P2P network overlay. They have also identified several fundamental operations that peers use for communication, including Join, Move, and Leave, along with protocols for managing game state among the P2P overlay. The authors have shown that these three procedures, within a Voronoi-based overlay,

provide a scalable solution as the number of peers increases. The project maintains a continually-updated, open source implementation of a P2P NVE library and simulation.

Two models that specify a hybrid P2P design are FreeMMG [14] by Kienzle, et al., and Peer Clustering [15] by Chen and Muntz. Both of these models incorporate the use of a managed server for secure storage of data, such as usernames, passwords, and long term object persistence. FreeMMG divides the world into segments and defines protocols for migrating objects as they move between segments. Peers subscribe to these segments to receive updates. The segments are managed by the server, which also manages each peer's area of interest. FreeMMG imposes a lockstep timing protocol, which precludes decentralized control or scalability. Peer Clustering divides the virtual environment into regions small enough for a peer to manage and it utilizes a distributed hash table approach for dividing the region responsibilities among active peers. Communication between peers is handled through the use of a Pastry-based routing scheme [9]. The conclusions and results presented by Chen and Muntz come from a simulated system model.

3. Model

This section introduces the current design of Audrey, our hybrid P2P system. The model specifies a managed server within a P2P framework, placing its design into the class of hybrid P2P systems. The server is responsible for account registration, login, logout, bootstrapping, long-term persistence, and a number of other tasks. It is very much a server with ultimate VE responsibility, albeit significantly lighter weight than the server in a typical client-server model. Peers are organized into a Voronoi-based overlay network, inspired by, but significantly expanding and detailing the procedures introduced by the Voronoi Overlay Network (VON) [11]. The Voronoi overlay is used to determine peer connectivity for communication and also for spatially decomposing the VE and distributing the simulation workload among its peers.

3.1 Server

The server's role is to provide a managed resource for hosting a secure NVE. The server manages all data requiring long-term persistence, such as account information, player characters, statistics, and persistent environment objects. Briefly noted, the following are the major components and services of the server: database management, account creation and maintenance, overlay bootstrapping, peer login, and peer logout. The server shares some services common to standard client-server NVE designs, deviating significantly by having the VE simulation shared among its active peers, rather than computing it itself, it does not participate in the VE simulation. This is the key distinction; the server is not a computational component.

Overlay bootstrapping is the process through which peers become connected to the network overlay. Peers initially login to the server and upon successful validation, receive session credentials. Following login, the peer makes a request to join the overlay, also known as forwarding. During an active session a peer is assigned an initial VE position, either by the server or from the peer overlay. Following login, the server acts as the initial point of contact for forwarding. Upon receipt of a forwarding request, the server responds by sending the peer the contact information of an active peer it believes is closest to the indicated starting location. The forwarding peer then contacts the active peer to continue the forwarding process; this repeats until the peer closest to the starting position is found. Upon completion, the peer initiates a neighborhood discovery protocol to reveal its peer neighbor environment and begins active participation in the overlay. Following initial forwarding, any time a new and distant VE position is indicated, an active peer already known to the forwarding peer can be used as the initial point of contact, bypassing the server entirely.

The Audrey model requires a server in order for overlay bootstrapping to be secure and reliable. It is not reasonable to rely upon a pure P2P framework that makes use of cached history of peers or a distributed hash table (DHT). In addition to security concerns in forming an overlay, when a peer leaves the overlay or unexpectedly fails for any reason, it can not be expected to be available for contact or overlay maintenance support. The Audrey model provides a guaranteed point of initial and fallback contact for overlay bootstrapping and peer validation.

3.2 Peer

All peers are homogeneous; there is no differentiation in function or responsibility among any of the peers. Peers coordinate among each other to maintain the VE simulation, interacting with the server only for account creation, login, initial forwarding, and logout. The peer process is heavyweight in that it has a wide range of computational responsibilities, including networking demands, simulation workload, and rendering of the VE from the player's perspective.

The Voronoi diagram is applied in two different contexts to form and maintain a VE simulation [16]. The first context describes the overlay connectivity among the peers for network communication. The second context spatially decomposes the VE into regions of responsibility among the peers. Each peer maintains a diagram based upon all peers with which it is interested, its neighbors. The edges shared between the peer region and other peers indicate direct or enclosing, with all others identified as area of interest (AOI) neighbors. The peer utilizes the data structure to know which other peers should receive updates on movement, new object creation, events. Additionally, it uses the data structure to know when to inform a neighboring peer of the arrival

of a new neighbor. The diagram is also used to identify regions of shared responsibility for objects having short-term persistence. In order to effectively support fault tolerance, the simulation of all objects within the VE is replicated and coordinated among neighboring peers. As new objects are created, the responsibility for their shared/replicated simulation is described by the diagram.

Peers have no need for routing of messages through the overlay when communicating with each other, Network communication between peers is always direct. Throughout a peer's active session, every peer they have ever had contact with is recorded and kept in memory, which includes the network endpoint for communication. Any time communication between peers is necessitated, a direct connection is made.

3.3 Security

Security is a core design element, supported primarily through the use of public/private key pairs for all peers, as well as the server. The server is assigned a public/private key pair; each peer is also assigned a public/private key pair upon account creation. Peers are distributed with the server's public key, which is used throughout the framework for verification without requiring that contact is made with the server. The public/private key pair for all peers is stored at the server and securely transmitted, through an SSL connection, during login. These keys form the basis for various signing and verification protocols used throughout the framework.

During login, the server generates a unique session certificate, signed by the server's private key. The certificate includes the peer's public key, networking endpoint, date the certificate was created, and the length of time the certificate is valid. The certificate's expiration serves to limit the time a rouge process can wreak havoc in the case a certificate is compromised. Upon initial contact, peers exchange certificates, enabling them to verify the authenticity of each other by verifying that the presented certificate's signature is from the server. It is this essential design element that allows peers to validate each other without having to contact the server, supporting the security and scalability of the framework.

4. Implementation

This section details the implementation of the Audrey model as a distributed application, following the model described in Section 3. The server and peer processes share significant design and source code components. Unless noted, the sub-sections below describe functionality common to both components. The development environment and tools are briefly highlighted, the spatial overlay technique is detailed, followed by a detailed description of the processing architecture, networking, security, data collection, and visualization components.

Audrey is an actual implementation of a VE model; it is not a simulation of one. While it is an operational

framework, it is spurious to suggest that Audrey is ready for a commercial, real-world deployment. Further technical hurdles remain to be overcome, along with the design and development of a massive virtual environment that is appropriate for the capabilities enabled this hybrid P2P framework.

4.1 Development Environment

The server and peer processes are written in C++, utilizing several third-party libraries, including Boost[17], Crypto++[18], RapidXML[19], and Google's gtest[20], along with a custom implementation of the Voronoi diagram data structure. Primary development is performed using Visual Studio in a Microsoft Windows environment, although a Linux makefile is maintained, enabling the framework to be compiled and executed on both Windows and Linux operating systems.

Boost is used to provide cross platform threading, cross platform data type definitions, and networking, which includes the SSL networking protocol used for the secure transmission of username, passwords, and public/private key pairs. The Crypto++ library provides the public/private key generation, signing, and verification functionality, along with other cryptographic capabilities. RapidXML provides XML parsing and persistence, which is used for the persistence of an XML based user account database for the server, as well as data collection persistence. The Google gtest library is incorporated into the project to support unit testing.

4.2 Spatial Overlay

The Audrey model specifies the use of a Voronoi diagram to form the peer overlay. The implementation of our model is supported through the use of a custom C++ implementation of the Voronoi diagram. The design of the overlay code is such that the nature of the spatial diagram is unknown throughout the application. The overlay behavior is defined in an abstract base class named *P2POverlay*, which details what a spatial diagram must support in order to work as an overlay. These behaviors include the ability to construct a diagram from a set of points and then perform different neighbor queries when given a specific point. The custom Voronoi implementation derives from this abstract base class and provides concrete implementations for these behaviors. In the future, if it is desired for any reason to use a different spatial overlay structure, there is a relatively straightforward process for doing so. The new data structure simply derives from the abstract *P2POverlay* class and the application instantiates the new data structure instead of the Voronoi diagram; the rest of the application code remains untouched.

4.3 Processing Architecture

The underlying processing design of the peer and server is that of a fan-out task scheduling and processing core. At startup, the process creates a thread pool with an initial worker thread count matching the number of available CPU

cores. These worker threads all listen to a single, synchronized work queue that signals an event upon receipt of a new work item. The design of an event ensures that only one inactive worker is signaled upon receipt of a new task. As it completes its current task, a worker checks to see if the work queue contains more work items. If it does, the worker grabs the next item and continues processing; otherwise it enters an efficient wait state, waiting to be signaled. This design and implementation enables a highly scalable computational framework for both the server and peer processes.

4.4 Networking

All network communication is over UDP, except for account creation and login, which is TCP over an SSL connection. Every communication that occurs between the server and a peer or between peers is performed within the context of a *protocol*. Each protocol is described as a state machine, one for the initiator and one for the receiver. The state machine describes the messages that are sent and received, along with the state transitions that occur upon the receipt of a message. The protocol also describes timeouts and retries for each state in support of messaging failures and/or delays. The following list identifies the different protocols supported by the current framework implementation:

- 1) Create Account
- 2) Login
- 3) Logout
- 4) Start Position
- 5) Alive
- 6) Forwarding
- 7) Neighborhood Discovery
- 8) New Neighbor
- 9) Move

The framework defines a base protocol class that provides the underlying implementation of the state machine, from which all protocols are derived. The base implementation provides the functionality for state processing, transitions, timeout periods, number of retries for each state, and a callback mechanism for reporting the completion of the protocol. The protocols are designed to fit within the processing architecture described in Section 4.3. A master table of all protocols is maintained. Upon receipt of a message, the protocol associated with that message is assigned to a task and the next available worker thread processes the task and updates the state of the protocol. When the processing of the task is complete, the worker thread returns to the thread pool. This design allows any protocol to be processed by any available worker thread, rather than dedicating a single thread for protocol's lifetime.

4.4.1 Punch Through NAT

The networking implementation of Audrey supports punch through NAT given the design of the server and session

certificates. Peers are distributed with the server's public IP address, enabling them to contact the server at startup. The server sees the peer's public IP address/port combination and this becomes the networking endpoint recorded in the session certificate. The certificate is used by peers as the source for networking endpoints in contacting each other. Therefore, all peers communicate with each other through their external IP address, as seen by the server, leaving routing up to local switches and routers.

4.4.2 Dropped Packets & Latency

Because the network the framework runs on is a best-case scenario (a local area network without external traffic; refer to Section 5), a small concession is made to simulate real-world Internet conditions of packet loss and latency. The code is designed with a receiving queue of messages, which is parameterized with the ability to drop or delay incoming packets according to a set of runtime parameters. This provides the system the ability to execute the framework under different levels of packet loss and latency throughout the network.

4.5 Security

As described in Section 3.3, the security model relies upon public/private key pairs, along with server signed certificates. The server is assigned unique public/private key, with its public key distributed to all peers, through a configuration file. The next few paragraphs highlight the use of key pairs and certificates throughout the framework.

An account certificate is created when a peer requests a new account. This certificate is composed of a unique account ID, username, password (SHA-256 hash), first name, last name, gameplay name, and a public/private key pair. The Crypto++ library is used to generate a public/private key pair and to perform an SHA-256 encoding of the password. During the framework development, a 1024 bit key size is used; this is a simple parameter that can be changed to suit whatever security needs exist for a real-world deployment. The account creation protocol is conducted using TCP over an SSL stream, provided by the Boost library, ensuring secure transmission of the username and password. The certificate is persisted at the server and never transmitted to any other entity in the framework.

Upon peer login, the server generates a session certificate. This certificate is composed of the peer's unique account ID, gameplay name, public key, date/time issued, date/time of expiration, and the peer's networking endpoint, all signed by the server's private key. During login the peer sends its username and password (SHA-256 hash), then receives its session certificate over an SSL connection to ensure secure transmission. This certificate is shared with other overlay peers at initial contact, allowing the authenticity of a peer to be validated without having to contact the server.

4.6 Data Collection & Visualization

The framework includes a separate post-execution data collection and visualization application. This capability is achieved through three different components. During execution, each peer maintains a detailed, in-memory, log of events, primarily network communications. Following execution of a VE scenario, the server contacts each peer in sequence, requests its log, and persists the received data into an XML file, one for each peer. The final component is a standalone application that provides the post-execution data visualization capabilities.

The standalone application is written in C#. It provides a user the ability to view raw data in tabular and graphical form. It also features the ability to replay an execution scenario, including support for pausing or stepping through event logs. The replay offers a global reconstruction of the overlay, along with the ability to view replay from the perspective of a single peer. This application also provides some data analysis features, based upon the global reconstruction, such as comparing the neighbors a peer saw versus the neighbors it should have seen. This ability to log the execution events, display, replay, and summarize those results in this application provides a significant step forward in the development and analysis of different techniques in the Audrey model.

5. Execution Platforms

Two different platforms are utilized to enable algorithm development and large-scale execution exercises. The first is a small local area network of 30 GuruPlug[21] computers, the second is a modest Beowulf[22] cluster of 128 CPU cores (Rex). The GuruPlug network is used as the primary application and algorithm development platform; it is dedicated solely to the Audrey research project. The Rex cluster is used for larger scale execution exercises.

The GuruPlug computers are small form factor computing devices. Their physical dimensions are approximately 2 x 3 x 4 inches. Each device includes a 1.2 GHz ARM based CPU running a Debian Linux 2.6 kernel, with 512MB of RAM, 512MB of flash storage, and a wired gigabit Ethernet connection. The GuruPlug computers run peer processes, with the server executing on a standard desktop computer, running either Windows or Linux. Each computer is easily capable of running three peer processes, enabling an execution scenario to utilize 90 asynchronous peer processes. When peer processes are created on this platform, they are given a unique configuration file to load, which includes their unique IP/port network address.

Rex is a Beowulf cluster; a small bit of custom code to support execution on this platform. This is distinct from the GuruPlug platform. Each node on Rex has two, quad core CPUs, totaling eight cores per node. Each node is assigned a unique, private NAT address. Peers created on this cluster

use the MPI API to obtain their unique process rank. Using a rank, a peer process forms a unique IP/port network address. This is the only use of MPI in the cluster environment, all networking is as previously described, using the socket API.

The Audrey framework is an actual implementation, not a simulation; all executions are performed in real-time. Elapsed execution scenario time is the same as the length of the execution; time is real, not simulated. The size of the cluster or number of computers in the network affects only the number of peer processes that can participate in the execution, not the length of time it takes to perform the execution.

6. Closing Remarks

This paper presents Audrey, an ongoing research project and development framework. The purpose of the Audrey project is the development of a comprehensive model, as well as the demonstration of a successful implementation of P2P techniques to enable massive P2P-based NVEs. At this point, the primary contributions of this effort are three-fold. The first is a model that combines a managed server, a Voronoi-based P2P overlay, and a security model to form a hybrid NVE design. The second is an operational implementation of the model that executes on physically distributed and asynchronous computing devices. The third is a post-execution application capable of visualizing, replaying and summarizing the real-time execution data logs.

References

- [1] B. Entertainment, "World of warcraft subscriber base reaches 12 million worldwide," Oct 2010. [Online]. Available: <http://us.blizzard.com/en-us/company/press/pressreleases.html?101007>
- [2] Gamasutra, "Gdc austin: An inside look at the universe of warcraft."
- [3] CCP, "Eve online," <http://www.eveonline.com>. [Online]. Available: <http://www.eveonline.com>
- [4] L. Fan, H. Taylor, and P. Trinder, "Mediator: a design framework for p2p mmogs," in *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA: ACM, 2007, pp. 43–48.
- [5] J. Keller and G. Simon, "Solipsis: A massively multi-participant virtual world," in *Parallel and Distributed Processing Techniques and Applications*. CSREA Press, 2003, pp. 262–268.
- [6] D. Frey, J. Royan, R. Piegay, A. Kermarrec, E. Anceaume, and F. Fessant, "Solipsis: A decentralized architecture for virtual environments," in *Workshop on Massively Multiuser Virtual Environments*, 2008.
- [7] D. Hughes, "Exploiting p2p in the creation of game worlds," in *In the proceedings of ACM GDTW 2005*, 2005.
- [8] B. Knutsson, M. M. Games, H. Lu, W. Xu, and B. Hopkins, "Peer-to-peer support for massively multiplayer games," 2004.
- [9] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. London, UK: Springer-Verlag, 2001, pp. 329–350.
- [10] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications (JSAC)*, vol. 20, p. 2002, 2002.

- [11] S.-Y. Hu and G.-M. Liao, "Scalable peer-to-peer networked virtual environment," in *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA: ACM, 2004, pp. 129–133.
- [12] J.-F. Chen, W.-C. Lin, T.-H. Chen, and S.-Y. Hu, "A forwarding model for voronoi-based overlay network," in *ICPADS '07: Proceedings of the 13th International Conference on Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1–7.
- [13] J.-R. Jiang, Y.-L. Huang, and S.-Y. Hu, "Scalable aoi-cast for peer-to-peer networked virtual environments," in *ICDCSW '08: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems Workshops*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 447–452.
- [14] F. R. Cecin, R. de Oliveira Jannone, C. F. R. Geyer, M. G. Martins, and J. L. V. Barbosa, "Freemmg: a hybrid peer-to-peer and client-server model for massively multiplayer games," in *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA: ACM, 2004, pp. 172–172.
- [15] A. Chen and R. R. Muntz, "Peer clustering: a hybrid approach to distributed virtual environments," in *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA: ACM, 2006, p. 11.
- [16] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Computing Surveys*, vol. 23, no. 3, pp. 345–405, 1991.
- [17] Boost.org, "Boost c++ libraries," <http://www.boost.org>. [Online]. Available: <http://www.boost.org>
- [18] Crypto++, "Crypto++ library," <http://www.cryptopp.com>. [Online]. Available: <http://www.cryptopp.com>
- [19] M. Kalicinski, "Rapidxml," <http://rapidxml.sourceforge.net>. [Online]. Available: <http://rapidxml.sourceforge.net>
- [20] I. Google, "googletest," <http://code.google.com/p/googletest>. [Online]. Available: <http://code.google.com/p/googletest>
- [21] Wikipedia, "Guruplug — wikipedia, the free encyclopedia," 2011, [Online; accessed 10-February-2011]. [Online]. Available: <http://en.wikipedia.org/w/index.php?title=GuruPlug&oldid=411780301>
- [22] —, "Beowulf (computing) — wikipedia, the free encyclopedia," 2011, [Online; accessed 12-February-2011]. [Online]. Available: [http://en.wikipedia.org/w/index.php?title=Beowulf_\(computing\)&oldid=413043434](http://en.wikipedia.org/w/index.php?title=Beowulf_(computing)&oldid=413043434)

Cycle Embedding in Folded Hypercubes

Y-Chuang Chen¹ and Lih-Yu Lin²

^{1,2}Department of Information Management, Ming Hsin University of Science and Technology,
Hsin Feng, Hsinchu 30401, Taiwan, R.O.C.

Abstract—The cycle embedding problem in interconnection networks is an important issue, because it is one of measurements for determining whether the topology of a network is suitable for an application in which embedding rings of various lengths into the topology is required. Embedding cycles of different sizes into a network are benefit to execute parallel programs efficiently. The folded hypercube is a popular network because of its attractive properties. Given an n -dimensional folded hypercube FQ_n and let e be any edge of FQ_n . In this paper, we discuss the number of simple k -cycles in FQ_n , which passing through the edge e , for $k = 4, 6$.

Keywords: folded hypercube, simple cycles, simple paths, cycle embedding.

1. Introduction

For the graph definition and notation, we follow [1]. $G = (V, E)$ is a simple graph if V is a finite set and E is a subset of $\{(u, v) | (u, v) \text{ is an unordered pair of } V\}$. We say that V is the vertex set and E is the edge set. The neighborhood of v , $N_G(v)$, is $\{x | (v, x) \in E(G)\}$. Two vertices a and b are *adjacent* if $(a, b) \in E(G)$. A *path* $P = \langle v_0, v_1, \dots, v_m \rangle$ is a sequence of adjacent vertices. A *cycle* $C = \langle v_0, v_1, \dots, v_m, v_0 \rangle$ is a sequence of adjacent vertices where the first vertex is the same as the last one. A path P (a cycle C , respectively) is a *simple path* (*simple cycle*, respectively) if $v_i \neq v_j$ for every $i \neq j$, otherwise it is a *non-simple path* (*non-simple cycle*, respectively). In this paper, we abbreviate a *simple path* and a *simple cycle* as a *path* and a *cycle*, respectively. The length of a path P (a cycle C , respectively), denoted by $\ell(P)$ ($\ell(C)$, respectively) is the number of edges in P (in C , respectively). A simple path (cycle, respectively) of length k , $k \geq 3$, is abbreviated as a *k -path* (*k -cycle*, respectively). The *distance* from vertex u to vertex v , denoted by $dist(u, v)$, is the minimum length of any path from u to v .

A graph G is *pancyclic* if any k -cycle can be embedded into it for $3 \leq k \leq |V(G)|$; G is *bipancyclic* if any k -cycle of even length can be embedded into it for $4 \leq k \leq |V(G)|$. A pancyclic graph G is *edge-pancyclic* if every edge of G lies on a cycle of every length; a bipancyclic graph G is *edge-bipancyclic* if every edge of G lies on a cycle of every even length. Bipancyclicity is essentially a restriction of the concept of pancyclicity to cycles of even lengths. Saad and Schultz [13] had proved that the hypercubes Q_n are

bipancyclic and Li et al. [11] had proved that Q_n are edge-bipancyclic. There are also many literatures discussing the pancyclic and edge-pancyclic related properties on specific networks, such as Möbius cubes [17], balanced hypercubes [16], folded hypercubes [15], enhanced hypercubes [9], and exchanged hypercubes [12].

The cycle embedding problem is one of the most popular research issues in interconnection networks, because it is an important measurement for determining whether the topology of a network is suitable for an application in which embedding rings of various lengths into the topology is required [10]. Moreover, embedding cycles of different sizes into a network are benefit to execute parallel programs efficiently [7]. There are many literatures which discussing cycle embedding of various lengths. In 1971, the numbers of non-simple k -cycles for $k = 3, 4$ and 5 in a graph G are counted by F. Harary et al. [8]. In 2003, H. L. Fu et al. counted the number of non-simple 6-cycles in a graph G [3]. Until 2007, G. G. Cash proposed a mathematically exact method for finding the number of non-simple k -cycles for $3 \leq k \leq |V(G)|$ in a graph G [2]. On specific networks, there are also some literatures discussing the number of non-simple k -cycles and k -paths, such as the low-density parity check codes [6] and star graphs [14]. For the n -dimensional hypercube Q_n , the number of k -cycles in Q_n , which passing through any edge $e \in E(Q_n)$ for $k = 4, 6$, and 8 , is discussed in [4]. In this paper, we discuss the number of k -cycles in the n -dimensional folded hypercube FQ_n , which passing through any edge $e \in E(FQ_n)$ for $k = 4, 6$.

The remainder of this paper is organized as follows. In Section 2, we state some important properties of hypercubes and folded hypercubes, and also give some notations. In Section 3, the numbers of cycles of lengths 4 and 6 in folded hypercubes are discussed. Section 4 gives concluding remarks and future works.

2. Preliminaries

The *hypercube* is a popular network because of its attractive properties, including regularity, symmetry, small diameter, strong connectivity, recursive construction, partitionability, and relatively low link complexity [7], [11], [13]. The *folded hypercube*, proposed by A. El-Amawy and S. Latifi [5], is a variation of the hypercube. The folded hypercube includes many good properties of the hypercube, such as vertex-symmetry and edge-symmetry. The folded

hypercube is also superior over the hypercube in some properties, such as shorter average distance, half diameter, and less delay in communication than the hypercube [5], [15].

The formal definition of an n -dimensional hypercube, denoted by Q_n , is given as follows. Each vertex v in Q_n can be distinctly labeled by an n -bit binary string, $v = v_n v_{n-1} \dots v_1$. For $1 \leq i \leq n$, we use v^i to denote the binary string $v_n \dots \bar{v}_i \dots v_1$. The Q_n consists of all n -bit binary strings representing its vertices. Two vertices u and v are adjacent if and only if $v = u^i$ with some i . An n -dimensional hypercube Q_n can be constructed from two identical $(n-1)$ -dimensional hypercubes, Q_{n-1}^0 and Q_{n-1}^1 , where $V(Q_{n-1}^0) = \{v_n v_{n-1} \dots v_1 | v_n = 0\}$ and $V(Q_{n-1}^1) = \{v_n v_{n-1} \dots v_1 | v_n = 1\}$. The vertex set of Q_n is $V(Q_n) = V(Q_{n-1}^0) \cup V(Q_{n-1}^1)$; and the edge set is $E(Q_n) = E(Q_{n-1}^0) \cup E(Q_{n-1}^1) \cup E_n$ where E_n is a set of edges connecting the vertices of Q_{n-1}^0 and Q_{n-1}^1 in a one to one fashion. That is, $E_n = \{(0v_{n-1}v_{n-2} \dots v_1, 1v_{n-1}v_{n-2} \dots v_1) | v_i \in \{0, 1\} \text{ for } 1 \leq i \leq n-1\}$.

An n -dimensional folded hypercube, denoted by FQ_n , is obtained from Q_n by adding a specific perfect matching between Q_{n-1}^0 and Q_{n-1}^1 . An n -dimensional folded hypercube FQ_n is also constructed from two identical $(n-1)$ -dimensional hypercubes, Q_{n-1}^0 and Q_{n-1}^1 , where $V(Q_{n-1}^0) = \{v_n v_{n-1} \dots v_1 | v_n = 0\}$ and $V(Q_{n-1}^1) = \{v_n v_{n-1} \dots v_1 | v_n = 1\}$. The vertex set of FQ_n is $V(FQ_n) = V(Q_{n-1}^0) \cup V(Q_{n-1}^1) = V(Q_n)$; and the edge set is $E(FQ_n) = E(Q_n) \cup X_n$ where X_n is a perfect matching between Q_{n-1}^0 and Q_{n-1}^1 , such that for every vertex $v = v_n v_{n-1} \dots v_1 \in V(FQ_n)$, $(v_n v_{n-1} \dots v_1, \bar{v}_n v_{n-1} \dots v_1) \in X_n$. In FQ_n , any edge belongs to X_n is called a *complement edge*, otherwise it is called a *hypercube edge*. Fig. 1 illustrates FQ_2 and FQ_3 . We need some more terms. Let $v = v_n v_{n-1} \dots v_1$ be a vertex of FQ_n , we use v' to denote vertex $\bar{v}_n v_{n-1} \dots v_1$ and use \bar{v} to denote vertex $\bar{v}_n v_{n-1} \dots v_1$. That is, (v, v') is a hypercube edge and (v, \bar{v}) is a complement edge of FQ_n . We notice that folded hypercubes are vertex-symmetric and edge-symmetric.

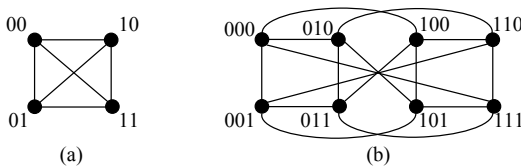


Fig. 1: (a) FQ_2 ; (b) FQ_3 .

To show the number of k -cycles in folded hypercubes, some definition are needed. Let $P_k(G)$ be a k -path in graph G , $P_k^e(G)$ be a k -path with edge e (called *required edge*) in graph G , and $ndp_k^e(G)$ be the number of k -paths $P_k^e(G)$ s in graph G . Also, let $C_k(G)$ be a k -cycle of length k in graph G , $C_k^e(G)$ be a k -cycle with edge e (also called *required edge*) in graph G , and $ndc_k^e(G)$ be the number of k -cycles

$C_k^e(G)$ s in graph G .

3. Number of k -cycles in folded hypercubes

To show the number of k -cycles with $k = 4, 6$ in folded hypercubes, the following four lemmas are essential.

Lemma 1: Given an n -dimensional hypercube Q_n for $n \geq 2$. For every edge $(u, v) \in E(Q_n)$, there exist exactly $n-1$ 3-paths joining u and v .

Proof. Since the hypercube Q_n is edge-symmetric and $dist(u, v) = 1$, we may without loss of generality assume that $(u, v) = (a_n a_{n-1} \dots a_1, a_n a_{n-1} \dots \bar{a}_1)$. Then the $n-1$ 3-paths joining u and v are as follows:

$$\langle a_n a_{n-1} \dots a_i \dots a_1, a_n a_{n-1} \dots \bar{a}_i \dots a_1, a_n a_{n-1} \dots \bar{a}_i \dots \bar{a}_1, a_n a_{n-1} \dots a_i \dots a_2 \bar{a}_1 \rangle, 2 \leq i \leq n. \quad \square$$

Lemma 2: Let n and k be two positive integers such that $n \geq k \geq 1$. Given an n -dimensional hypercube Q_n . For every pair of $u, v \in V(Q_n)$ such that $dist(u, v) = k$, there exist exactly $k!$ k -paths joining u and v .

Proof. Since $dist(u, v) = k$, the labels of vertices u and v are different with exactly k bits. For every k -path joining u and v , the labels of every two consecutive vertices of this path are different with only one of the k bits. Hence, there are exactly $k!$ k -paths joining u and v and the lemma is proved. \square

Lemma 3: Given a 3-dimensional folded hypercube FQ_3 . For every edge $e \in E(FQ_3)$, $ndc_4^e(FQ_3) = 9$.

Proof. Since folded hypercubes are edge-symmetric, we may without loss of generality assume that the required edge $e = (x, y) = (000, 001)$. Then, the 9 4-cycles $C_4^e(FQ_3)$ s are as the following by brute force:

$$\begin{aligned} \langle 000, 001, 011, 010, 000 \rangle, & \quad \langle 000, 001, 011, 100, 000 \rangle, \\ \langle 000, 001, 011, 111, 000 \rangle, & \quad \langle 000, 001, 101, 010, 000 \rangle, \\ \langle 000, 001, 101, 100, 000 \rangle, & \quad \langle 000, 001, 101, 111, 000 \rangle, \\ \langle 000, 001, 110, 010, 000 \rangle, & \quad \langle 000, 001, 110, 100, 000 \rangle, \\ \langle 000, 001, 110, 111, 000 \rangle. & \quad \square \end{aligned}$$

Lemma 4: [4] Given an n -dimensional hypercube Q_n for $n \geq 3$. Let e be any edge in Q_n , then $ndc_6^e(Q_n) = 4n^2 - 12n + 8$.

For the number of 4-cycles and 6-cycles with any required edge in folded hypercubes, it is proved in Theorems 1 and 2.

Theorem 1: Given an n -dimensional folded hypercube FQ_n for $n \geq 4$. For every edge $e \in E(FQ_n)$, $ndc_4^e(FQ_n) = n$.

Proof. Since folded hypercubes are edge-symmetric, we may without loss of generality assume that the required edge

$e = (v_n v_{n-1} \dots v_1, v_n v_{n-1} \dots \bar{v}_1)$. Then, the n 4-cycles are as follows:

- (1) $\langle v_n v_{n-1} \dots v_i \dots v_1, v_n v_{n-1} \dots \bar{v}_i \dots v_1, v_n v_{n-1} \dots \bar{v}_i \dots \bar{v}_1, v_n v_{n-1} \dots v_i \dots \bar{v}_1, v_n v_{n-1} \dots v_i \dots v_1 \rangle$, $2 \leq i \leq n$; and
- (2) $\langle v_n v_{n-1} \dots v_i \dots v_1, \bar{v}_n v_{n-1} \dots v_i \dots v_1, \bar{v}_n v_{n-1} \dots v_i \dots \bar{v}_1, v_n v_{n-1} \dots v_i \dots \bar{v}_1, v_n v_{n-1} \dots v_i \dots v_1 \rangle$.

Therefore, there are totally $(n-1) + 1 = n$ $C_4^e(FQ_n)$ s and the theorem is proved. \square

Theorem 2: Give an n -dimensional folded hypercube FQ_n for $n \geq 6$. For every edge $e \in E(FQ_n)$, $ndc_6^e(FQ_n) = 4n^2 - 4n$.

Proof. Since folded hypercubes are edge-symmetric, we may without loss of generality assume that the required edge $e = (x, y) = (0a_{n-1} \dots a_1, 0a_{n-1} \dots \bar{a}_1)$. Hence, (x, y) is in Q_{n-1}^0 . Let $SP_0 = C_6^e(FQ_n) \cap Q_{n-1}^0$ and $SP_1 = C_6^e(FQ_n) \cap Q_{n-1}^1$. According to the number of edges of $C_6^e(FQ_n) \cap X_n$, we divide the proof into four cases.

Case 1: $|C_6^e(FQ_n) \cap X_n| = 0$. By Lemma 4, $ndc_6^e(Q_n) = 4n^2 - 12n + 8$. Therefore, there are $4n^2 - 12n + 8$ 6-cycles in FQ_n in this case.

Case 2: $|C_6^e(FQ_n) \cap X_n| = 1$. Then, $C_6^e(FQ_n) \setminus X_n$ is a 5-path. Note that each edge of this path $C_6^e(FQ_n) \setminus X_n$ is a hypercube edge and the labels of the two endpoint vertices of this path are different with exactly n bits, which is a contradiction for $n \geq 6$. Therefore, it is impossible that $|C_6^e(FQ_n) \cap X_n| = 1$ for $n \geq 6$.

Case 3: $|C_6^e(FQ_n) \cap X_n| = 2$. Then, $|C_6^e(FQ_n) \cap E_n| = 0$ or 2. There are totally $(8n - 16) + 8 = 8n - 8$ 6-cycles in Case 3 as shown in the following two subcases.

Case 3-1: $|C_6^e(FQ_n) \cap E_n| = 0$. Hence, SP_0 and SP_1 are a path of Q_{n-1}^0 and Q_{n-1}^1 , respectively, and $\ell(SP_0) + \ell(SP_1) = 4$. There are totally $(3n - 6) + (4n - 8) + (n - 2) = 8n - 16$ 6-cycles in Case 3-1 as shown in the following three subcases.

Case 3-1-1: $\ell(SP_0) = 3$ and $\ell(SP_1) = 1$. By Theorem 1, $ndc_4^e(Q_{n-1}^0) = n - 2$. Let one of the $C_4^e(Q_{n-1}^0)$ s be $\langle x, y, a, b, x \rangle$. Since $|C_6^e(FQ_n) \cap X_n| = 2$, $\langle x, y, a, b, x \rangle$ can be extended to the three $C_6^e(FQ_n)$ s: $\langle x, y, a, b, \bar{b}, \bar{x}, x \rangle$, $\langle x, y, a, \bar{a}, \bar{b}, b, x \rangle$ and $\langle x, y, \bar{y}, \bar{a}, a, b, x \rangle$. Therefore, there are totally $(n - 2) \times 3 = 3n - 6$ $C_6^e(FQ_n)$ s in Case 3-1-1.

Case 3-1-2: $\ell(SP_0) = 2$ and $\ell(SP_1) = 2$. Since Q_{n-1}^0 is $(n - 1)$ -regular and contains no triangle, $|(N_{Q_{n-1}^0}(x)) \cup (N_{Q_{n-1}^0}(y)) \setminus \{x, y\}| = 2(n - 2) = 2n - 4$. So there are exactly $2n - 4$ 2-paths with edge (x, y) in Q_{n-1}^0 . That is, $ndp_2^e(FQ_{n-1}^0) = 2n - 4$. Let one of the $P_2^e(FQ_{n-1}^0)$ s be $\langle a, b, c \rangle$. Note that (x, y) is equal to either (a, b) or (b, c) . Since $\ell(SP_0) = \ell(SP_1) = 2$ and FQ_n contains no triangle for $n \geq 3$, $dist(a, c) = dist(\bar{a}, \bar{c}) = 2$. By Lemma 2, there are two 2-paths joining \bar{a} and \bar{c} in Q_{n-1}^1 , and let one of them be $\langle \bar{a}, d, \bar{c} \rangle$. Then, $\langle a, b, c, \bar{c}, d, \bar{a}, a \rangle$ forms a $C_6^e(FQ_n)$. As a result, there are totally $(2n - 4) \times 2 = 4n - 8$ $C_6^e(FQ_n)$ s in Case 3-1-2.

Case 3-1-3: $\ell(SP_0) = 1$ and $\ell(SP_1) = 3$. By Lemma 1, there are exactly $n - 2$ 3-paths joining \bar{x} and \bar{y} in Q_{n-1}^1 . Let one of the $n - 2$ paths be $\langle \bar{x}, a, b, \bar{y} \rangle$. Then, $\langle x, y, \bar{y}, b, a, \bar{x}, x \rangle$ forms a $C_6^e(FQ_n)$. Consequently, there are totally $n - 2$ $C_6^e(FQ_n)$ s in Case 3-1-3.

Case 3-2: $|C_6^e(FQ_n) \cap E_n| = 2$. There are totally $1 + 1 + 3 + 3 = 8$ 6-cycles in Case 3-2 as shown in the following four subcases.

Case 3-2-1: $\{(x, x'), (y, y')\} \subset E(C_6^e(FQ_n))$. There is only one 6-cycle with (x, y) : $\langle x, y, y', \bar{y}', x', x \rangle$.

Case 3-2-2: $\{(x, \bar{x}), (y, \bar{y})\} \subset E(C_6^e(FQ_n))$. There is also only one 6-cycle with (x, y) : $\langle x, y, \bar{y}, \bar{y}', \bar{x}', \bar{x}, x \rangle$.

Case 3-2-3: $\{(x, x'), (y, \bar{y})\} \subset E(C_6^e(FQ_n))$. There exist three 6-cycles with (x, y) : $\langle x, y, \bar{y}, \bar{y}', x', x \rangle$, $\langle x, y, \bar{y}, \bar{y}', y', x', x \rangle$, and $\langle x, y, \bar{y}, \bar{x}, x', x \rangle$. See Fig. 2.

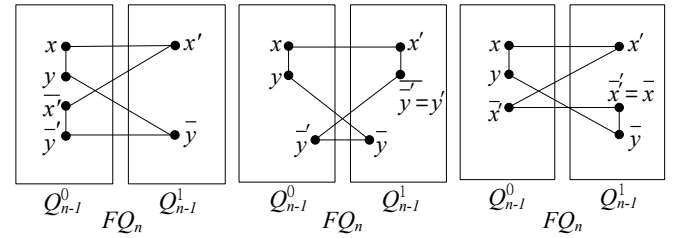


Fig. 2: $\{(x, x'), (y, \bar{y})\} \subset E(C_6^e(FQ_n))$.

Case 3-2-4: $\{(x, \bar{x}), (y, y')\} \subset E(C_6^e(FQ_n))$. There exist three 6-cycles with (x, y) : $\langle x, y, y', \bar{y}', \bar{x}', \bar{x}, x \rangle$, $\langle x, y, y', x', \bar{x}', \bar{x}, x \rangle$, and $\langle x, y, y', \bar{y}', \bar{y}, \bar{x}, x \rangle$. See Fig. 3.

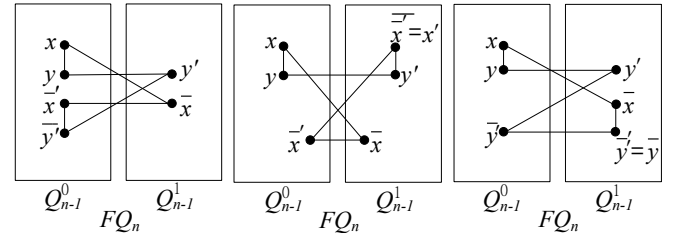


Fig. 3: $\{(x, \bar{x}), (y, y')\} \subset E(C_6^e(FQ_n))$.

Case 4: $|C_6^e(FQ_n) \cap X_n| \geq 3$. It is clear that it is impossible that $|C_6^e(FQ_n) \cap X_n| \geq 3$.

With above Case 1 ~ Case 4, $ndc_6^e(FQ_n) = (4n^2 - 12n + 8) + 0 + (8n - 8) + 0 = 4n^2 - 4n$ for $n \geq 6$, and the theorem is proved. \square

For the numbers of 6-cycles of small dimensional folded hypercubes FQ_n for $n = 3, 4, 5$, they can be obtained by computer programs and the results are stated in Lemmas 5~7.

Lemma 5: Given a 3-dimensional folded hypercube FQ_3 . For every edge $e \in E(FQ_3)$, there exist exactly 36 6-cycles with edge e in FQ_3 .

Lemma 6: Given a 4-dimensional folded hypercube FQ_4 . For every edge $e \in E(FQ_4)$, there exist exactly 48 6-cycles with edge e in FQ_4 .

Lemma 7: Given a 5-dimensional folded hypercube FQ_5 . For every edge $e \in E(FQ_5)$, there exist exactly 200 6-cycles with edge e in FQ_5 .

4. Conclusions

In this paper, we investigate in discussing embedding of distinct simple 4-cycles and 6-cycles with any required edge into folded hypercubes. For the number of 4-cycles with any required edge e in FQ_n , we show that $ndc_4^e(FQ_3) = 9$ and $ndc_4^e(FQ_n) = n$ for $n \geq 4$. For the number of 6-cycles with any required edge e in FQ_n , we show that $ndc_6^e(FQ_3) = 36$, $ndc_6^e(FQ_4) = 48$, $ndc_6^e(FQ_5) = 200$, and $ndc_6^e(FQ_n) = 4n^2 - 4n$ for $n \geq 6$. In the future, we hope to obtain $ndc_k^e(FQ_n)$ for every even $k \geq 8$.

Acknowledgments

This work was supported in part by the National Science Council of the Republic of China under Contract NSC 99-2221-E-159-014; Correspondence to: Y-Chuang Chen.

References

- [1] J. A. Bondy and U. S. R. Murty, Graph theory with applications, North Holland, New York, (1980).
- [2] Gordon G. Cash, "The number of n -cycles in a graph," *Applied Mathematics and Computation* 184, pp. 1080-1083 (2007).
- [3] Y. C. Chang and H. L. Fu, "The number of 6-cycles in a graph," *Bulletin of the ICA* 39, pp. 27-30 (2003).
- [4] Y. C. Chen and T. F. Jhang, "Embedding of simple cycles in hypercubes," accepted by the 2011 3rd International Conference on Computer Engineering and Technology (ICCET 2011).
- [5] A. El-Amawy and S. Latifi, "Properties and performance of folded hypercubes," *IEEE Transactions on Parallel and Distributed Systems* 2, pp. 31-42 (1991).
- [6] J. Fan and Y. Xiao, "A method of counting the number of cycles in LDPC codes," *International Conference of Signal Processing* 3, pp. 2183-2186 (2006).
- [7] F. F. Fang, "The bipanconnectivity of the hypercube," *Information Sciences* 178, pp. 4679-4687 (2008).
- [8] F. Harary and B. Manvel, "On the number of cycles in a graph," *Matematický časopis* 21, pp. 55-63 (1971).
- [9] L. Hongmei, "Cycles in enhanced hypercube networks," *International Seminar on Future Information Technology and Management Engineering*, pp. 560-563 (2008).
- [10] S. Y. Hsieh and J. Y. Shiu, "Cycle embedding of augmented cubes," *Applied Mathematics and Computation* 191, pp. 314-319 (2007).
- [11] T. K. Li, C. H. Tsai, Jimmy J. M. Tan, and L. H. Hsu, "Bipanconnectivity and edge-fault-tolerant bipanconnectivity of hypercubes," *Information Processing Letters* 87, pp. 107-110 (2003).
- [12] M. Ma and B. Liu, "Cycles embedding in exchanged hypercubes," *Information Processing Letters* 110, pp. 71-76 (2009).
- [13] Y. Saad and M. H. Schultz, "Topological properties of hypercubes," *IEEE Transactions on Computers* 37 (7), pp. 867-872 (1988).
- [14] X. Wu, M. He, F. Wang, J. Yang, and S. Latifi, "Distinct paths for the star graph," *International Conference on Communications and Mobile Computing*, pp. 322-326 (2009).
- [15] J. M. Xu and M. Ma, "Cycles in folded hypercubes," *Applied Mathematics Letters* 19, pp. 140-145 (2006).
- [16] M. Xu, X. D. Hu, and J. M. Xu, "Edge-pancyclicity and Hamiltonian laceability of the balanced hypercubes," *Applied Mathematics and Computation* 189, pp. 1393-1401 (2007).
- [17] M. Xu and J. M. Xu, "Edge-pancyclicity of Möbius cubes," *Information Processing Letters* 96, pp. 136-140 (2005).

A Cluster-Based Quantitative Reliability Model

Eduardo Cañete, Manuel Díaz, Luis Llopis and Bartolomé Rubio
 Dpto. Lenguajes y Ciencias de la Computación
 University of Málaga
 (ecc,mdr,luisll,tolo)@lcc.uma.es

Abstract—It is well known that Wireless Sensor and Actor Networks are error-prone as multi-hop communications are carried out. Furthermore, the further the distance between two nodes is, the less the communication reliability is. Despite the fact that this issue has been studied in many publications, there are new publications still appearing due to the importance of this topic. In this paper, we present a tool to help developers to better understand how the distance and the link qualities estimation affect the communication reliability between two nodes. We also present a reliability model to improve the reliability between nodes taking into account their energy consumption. The main feature of our proposal is that developers will be able to specify the desired reliability level quantitatively. Finally, a set of tests are carried out in order to study the performance of the proposed model.

Index Terms—Wireless Sensor and Actor Networks, Reliability, Model, Clusters

I. INTRODUCTION

Wireless Sensor and Actor Networks (WSANs) [1] are a promising technology which allows the monitoring and control of any kind of scenario (indoor environment, whole cities, woods, ...) [2]. These networks are composed of tiny devices which are resource-constraints enough due to their small size. They are normally characterized by their short-range wireless communications capabilities, short battery-lives, few memory and limited CPU processing capabilities. In spite of these limitations making WSAN applications difficult to develop because there exists another problem which is even worse. Within a WSAN, the delivery data between nodes (sensor and/or actors) which are N hops away from each other fail quite a lot due to the fact that WSANs are error-prone [3]. And obviously, the probability of fails increases if the distance between the source and destination also increases. Developers should take this into account this issue when they plan to develop and deploy a WSAN, otherwise the network probably does not achieve the goal for which it was thought. Thus, we can conclude that a very important issue in WSANs is to define efficient reliable multi-hop protocols in order to achieve either a high packet delivery probability (PDP) or a high packet reception ratio (PRR) [4].

In this paper, we present a tool developed to study how the delivery data between two nodes is affected as the distance (number of intermediate nodes) between them increases. But, the main contribution of this paper is a reliability model which allows the developers to numerically (0 to 100%) set the desired reliability level between two nodes which are N hops away from each other. Basically, the algorithm

is able to know and achieve the needed reliability of the intermediate nodes used to send information between nodes with a reliability previously specified by the user. This way of defining the desired reliability allows us to establish a more direct relationship between PRR and the application layer of our goal application. So, if we want to develop a WSAN application which is able to detect dangerous situations (for example, a high level of radiation), the sensor networks have to be capable of transmitting this kind of information with a reliability close to 100%. Other approaches allow us to set parameters such as high-reliability, medium-reliability, but what exactly is the meaning of these parameters? In other words, what exactly is the reliability reached by using the high-reliability or medium-reliability parameter? What about if we would like to establish a reliability lower than high-reliability and higher than medium-reliability? We think this way of defining the reliability levels among nodes confuses the developers.

The rest of the paper is structured as follows. Section II summarizes the related work. Section III describes the communication model on which the proposed reliability model is based. Section IV presents the proposed quantitative reliability model. Section V describes the reliability tool developed. Section VI discusses the performance evaluation of the proposed model. Finally, section VII concludes the paper.

II. RELATED WORK

There are different kinds of approaches focused on achieving reliable mechanisms to transport the data. Many of these approaches are designed for sensor networks where any scheme to organize the nodes is followed. In [5] is followed a reliability scheme based on the priority queues and work load of the nodes which allows the nodes to estimate the feasibility of delivering a packet on time. However, they impose important restrictions in the test scenario such as each node is assumed to know its position and only sensor-actor interaction is studied. Other reliable approach is established in [6]. In this work, packets are managed depending on the importance of their content, however the authors do not provide any algorithm to obtain the packet importance. In this approach, the packets are transmitted through different paths in order to increase the possibilities of data reception at destination. However, they assume there are not collisions and that packets are not cached in sensor nodes because of memory constraints. This last assumption can lead to a considerable increase of the overhead. In our approach, the reliable communication is

reduced to the clusters which make the protocol not only more energy-efficient but also robustness. In addition, our approach takes into account the existence of possible collisions and, on the other hand, the packets are registered in order to avoid an increase of the traffic overhead. It is worth pointing out, most of the current reliable protocols are designed to find the most reliable paths taking into account the energy consumption [7], [8], [9]. Of course, these kinds of approaches achieve good reliability levels and a good trade-off between reliability and energy consumption during the data transmissions but they are not able of quantifying the level of reliability that can be achieved as they are normally not based on a mathematical model. Furthermore, most of them are only designed to achieve reliable paths from the sensor nodes to the sink, cluster-head or base station, but not in an opposite way. In [10], authors propose a reliability model slightly similar to our model. They also allow developers to specify the desired reliability level between two nodes. To achieve that, they exploit the inherent redundancy of dense sensor networks by realizing probabilistic multi-path forwarding. In addition, they assume that all nodes have the same link qualities and the nodes know where are located geographically by using GPS coordinates. In contrast, our protocol is able to achieve the same goal without having to assume the constraints above mention (dense sensor network, same link qualities and geographical position).

III. COMMUNICATION MODEL

Most sensor networks are designed to transmit information from sensor nodes to one powerful node called sink, base station or cluster-head as the topology of the network. For example, we could have a network organized in several clusters within which all collected information is sent to the cluster-head which form another cluster whose cluster-head is the sink or the base station. The proposed reliability model thought, is to be implemented in networks which follow the aforementioned topology taking into account the following assumptions:

- 1) The communication pattern is many-one and one-many. There are groups of sensor nodes which transmit information to their leader node (cluster-head, sink, or base station). Leader nodes can also send information to the members of their groups. Although this kind of communication is less frequent, but not any less important.
- 2) Nodes are organized in levels (distance in hops to their leader node). So, each node knows what its level is regarding its leader node. This is known as gradient-based routing.
- 3) Nodes situated in level L also know their neighbors located in levels $L - 1$, $L + 1$ and L . Thanks to this information sensor nodes will know what the shortest paths are to reach their leader node as well as knowing how many hops there are to it. On the other hand, leader nodes have to send the information by using broadcast as they do not know where the member nodes

are located. However, they know in which level packet retransmission has to be interrupted.

- 4) Nodes know the link quality estimation between themselves and their neighbors.

IV. RELIABLE TRANSMISSION MODEL

As mentioned in previous section, it is assumed the nodes within a same cluster know in which level they are located and which are the different and shortest paths to send information to their cluster-head. To know the node level is equivalent to knowing the number of hops between this node and its cluster-head. Let us suppose that we have deployed the following lineal sensor network: 1-2-3-4-5 where 1 is the cluster-head, 5 is a sensor node and 2,3 and 4 are the intermediate nodes. The PDP of sending a packet from node 5 to the node 1 and viceversa comes defined by the product of the intermediates PDP as the following expressions show:

$$PDP_{51} = PDP_{54} * PDP_{43} * PDP_{32} * PDP_{21}$$

$$PDP_{15} = PDP_{12} * PDP_{23} * PDP_{34} * PDP_{45}$$

To make the discussion easier, PG_{ij} will be the PDP between nodes i and j when they are not neighboring and PC_{ij} when they are. Therefore, expressions above can also be expressed in the following way:

$$PG_{51} = PC_{54} * PC_{43} * PC_{32} * PC_{21} \quad (1)$$

$$PG_{15} = PC_{12} * PC_{23} * PC_{34} * PC_{45} \quad (2)$$

It is noteworthy that $PG_{51} \neq PG_{15}$ due to RSSI asymmetry. While RF theory states that the two directions of RF propagation have identical attenuation, in practice this is not the case [11].

These equations 1 and 2 show that to achieve a specific reliability (for example about 90%) during the transmission of packets from node 5 to node 1 (PG_{51}) it is also necessary to know a priori what the reliability is of the intermediate communications (PC_{54} , PC_{43} , PC_{32} , PC_{21}) which is quite hard due to the fact that the quality of each link changes in a dynamic and independent way over time. It implies we have to deal with an equation of $X - 1$ variables, where X is the number of nodes that participate in the communication process. Therefore, our first goal is to achieve that the equation used to calculate PG_{51} has only one variable. If all PC_{ij} were equals, we would have just one variable and:

$$PG_{ij} = PC^L$$

where L is the level of the source node i . From this equation we can find the value of PC in the following way:

$$\begin{aligned} PG_{ij} &= PC^L \\ \sqrt[L]{PG_{ij}} &= \sqrt[L]{PC^L} \\ \boxed{PC} &= \boxed{PG_{ij}^{\frac{1}{L}}} \end{aligned} \quad (3)$$

Continuing with our own example, equation 3 means that if we want to establish a reliability level of PG_{51} , the intermediate PC_{54} , PC_{43} , PC_{32} and PC_{21} must be equals to $PG_{51}^{\frac{1}{4}}$.

At this point, we know what must be the needed reliability level (PC) during the communication of the intermediate nodes to reach a specific reliability (PG_{ij}) between two nodes (i and j) which communication distance is L hops. Once, we know this information, the next step is to find the way of increasing PC_{ij} to $PG_{ij}^{\frac{1}{L}}$. It is obvious that if several retransmission are carried out from node i to node j the reliability between them will increase, but how many retransmissions are necessary to increase this reliability level from PC_{ij} to $PG_{ij}^{\frac{1}{L}}$? The following expression gives us the solution to this question:

$$\begin{aligned}
 1 - (1 - PC_{ij})^{N_{ij}} &= PG_{ij}^{\frac{1}{L}} \\
 -(1 - PC_{ij})^{N_{ij}} &= (PG_{ij}^{\frac{1}{L}}) - 1 \\
 (1 - PC_{ij})^{N_{ij}} &= 1 - (PG_{ij}^{\frac{1}{L}}) \\
 \ln(1 - PC_{ij})^{N_{ij}} &= \ln(1 - PG_{ij}^{\frac{1}{L}}) \\
 N_{ij} * \ln(1 - PC_{ij}) &= \ln(1 - PG_{ij}^{\frac{1}{L}}) \\
 N_{ij} &= \left\lceil \frac{\ln(1 - PG_{ij}^{\frac{1}{L}})}{\ln(1 - PC_{ij})} \right\rceil \quad (4)
 \end{aligned}$$

In the equation, $(1 - PC_{ij})^{N_{ij}}$ is the probability that node j does not receive a packet from node i after it is sent N times. Thus, $1 - (1 - PC_{ij})^{N_{ij}}$ is the probability that at least one packet sent by the node i arrives to the node j.

Basically, the communication reliable protocol is based on equations 3 and 4. For example, let us suppose that a developer has to create an application where nodes must send an alarm packet to the sink when they detect a high temperature (over a given threshold). If the furthest distance from them to the sink is four hops and the desired reliability is around 92%, equation 3 shows us that the link reliability between intermediate nodes must be $0.92^{\frac{1}{4}}$ which is 0.979 (about a 98%). Now, let us assume that the values of PC_{54} , PC_{43} , PC_{32} , PC_{21} are 78%, 85%, 88% and 81% respectively. Then, according to equation 4, node 5 needs to transmit the same packet to node 4 at least 3 times which comes from $\left\lceil \frac{\ln(1-0.98)}{\ln(1-0.78)} \right\rceil$. N_{43} , N_{32} and N_{21} would be equal to 2, 3 and 3 respectively.

This protocol depends heavily on the estimation of the current reliability between neighboring nodes. Thus, the more accurate the link quality estimation between neighboring nodes is, the better the achieved reliability between nodes which are far away from each others N hops is.

V. RELIABILITY TOOL

In order to help the developers understand what is going to be the impact of their established reliability levels. We have developed a tool (see figure 1) to help them analyze how the different reliability levels affect to the sensor networks depending on the number of levels established within a cluster and the desired reliability level in a multi-hop communication. The tool graphic interface can be classified in 4 parts:

1) Multi-hop communication parameters. This part is located in the top left corner of the interface and has the following elements:

- A numeric field which allows us to introduce the number of levels of the cluster where a reliable communication is going to be carried out. It also means, the maximum number of hops needed to send information from the sensor nodes to its cluster-head or viceversa.
- A numeric field where the desired level of reliability is indicated.
- A label which indicates to us the needed reliability during the communication of the intermediate nodes to achieve the global reliability specified.

2) Information Zone. It is located in the top middle of the interface. It just shows us the information mentioned above in a graphical way.

3) Single-hop communication parameters. It is located in the top right corner of the screen and it only has two fields:

- A numeric field which simulate the possible reliability current levels between two nodes.
- A numeric field which indicates to us the needed retransmissions number to achieve the reliability level calculated and is showed in the top left corner of the screen knowing that the current reliability between two nodes is the value established in the above field.

4) Graphic information. It is formed by four graphics through which it is easier to analyze how the application will perform depending on the established parameters.

a) Retransmissions Vs. Current Reliability. This graph is obtained from equation 4. It shows us how many retransmissions are needed to achieve the desired global reliability depending on the possible current reliabilities between neighboring nodes. For example, if we want to achieve a global reliability about 80% between two nodes which are 8 hops away from each other, equation 3 indicates that the required reliability per link is a 98%. The graphic shows us the number of retransmissions each node would need to achieve a global reliability of 80% on the basis of their current reliabilities. If we want to study a concrete data, the graph has a horizontal red line that shows us this kind of information. This line can be moved by modifying the data referred to the single hop communication (top right corner of screen). In our own example, the red line shows us that if a reliability between two nodes is about 20%, 17 retransmissions are needed to achieve a reliability level of 98% which is necessary to achieve the global reliability of 80%.

b) Derivative. It shows us the derivative of the previous graphic. Thanks to this graph it is possible to analyze what is the point from which the number of needed retransmissions goes up exponentially. Therefore, this graphic shows developers that when

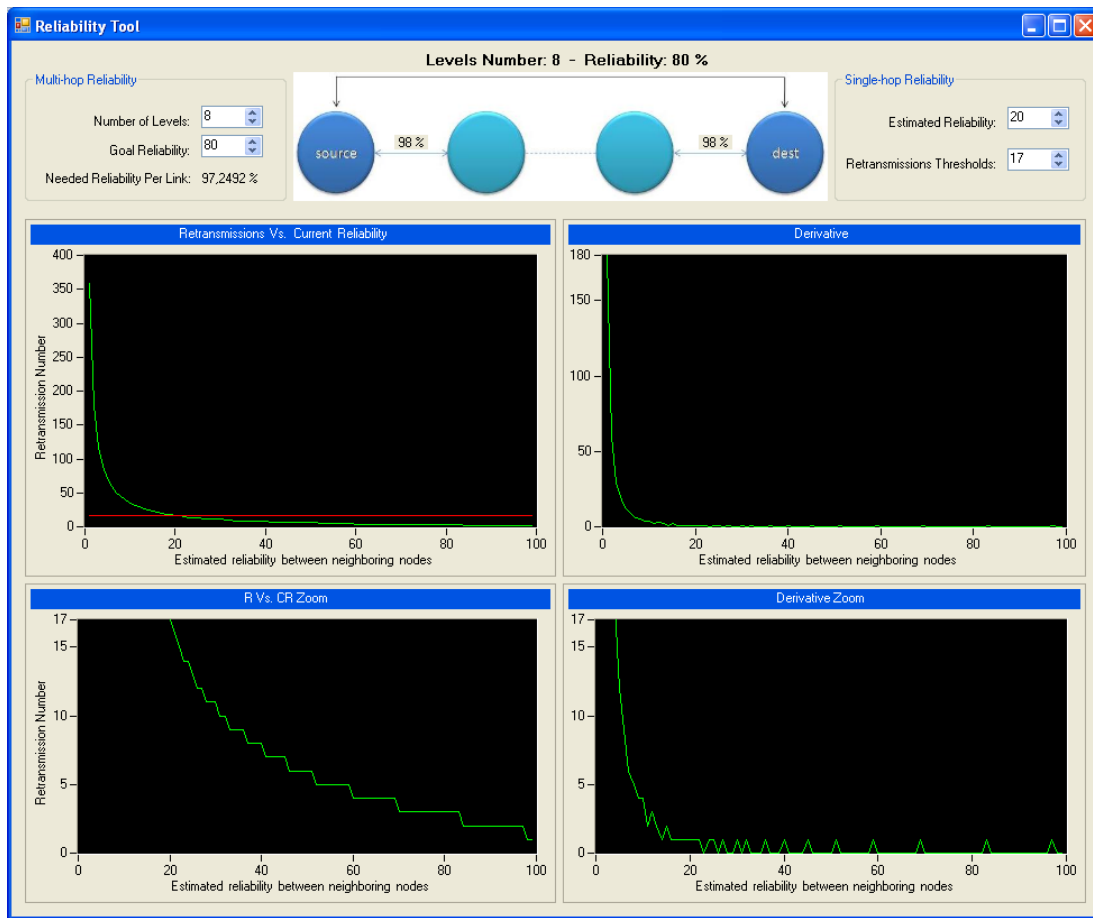


Fig. 1. Reliability Tool

the estimated reliability between two nodes is less than 20%, it is not advisable to retransmit a packet N times since N could be so big that the energy consumption of the nodes would be too costly.

- c) R Vs. CR Zoom. It is a zoom of the “Retransmissions Vs. Current Reliability” graphic. The zoom is established on the basis of the red line mentioned above.
- d) Derivative Zoom. This graphic shows us an interesting piece of information. Let us focus on the range 60-80, we can observe two plain signals in the subranges 60-70 and 72-80. This means that when estimated reliability between two nodes fall into one of these ranges, the number of required retransmissions is equal and therefore, the energy consumption is also the same. Developers may think that is is more costly (in terms of energy consumption) to achieve a reliability of 98% when the estimated reliability is 60% than to achieve the same when the estimated reliability is 70%. This graph reveals the ranges where energy consumption is the same independently wether the value of the estimated reliability is higher or lower.

On the other hand, this information allows the protocol to be more efficient at distributing the energy consumption over the whole network.

Figure 2 shows another different way of analyzing and understanding the relation between link quality among neighbors, the number of retransmissions to increase these link qualities and the desired goal reliability between two nodes which are L hops away from each other. The data represented in the figure have been generated by using the equation 4 and taking into account that L is equal to 9. In order to understand the data, let us focus on the gray area of the figure. For example, let us assume that a developer wants to achieve a reliability about a 90%. The figure indicates that if the estimated link qualities of the nodes are about a 89%, 77% or 67%, the protocol will need 2, 3 or 4 retransmissions respectively to achieve the desired reliability goal. Now, let us imagine that the estimated link quality of two nodes is greater than 67% and lower than 77%. In this case, the number of necessary retransmissions will have decimals (3.3, 3.4, ...). Thus, in order to ensure that the reliability goal (90%) is achieved, protocol will use the next integer. In this particular case, it would be the number 4. It could cause the final reliability goal is greater than 90%. We have considered that it is better

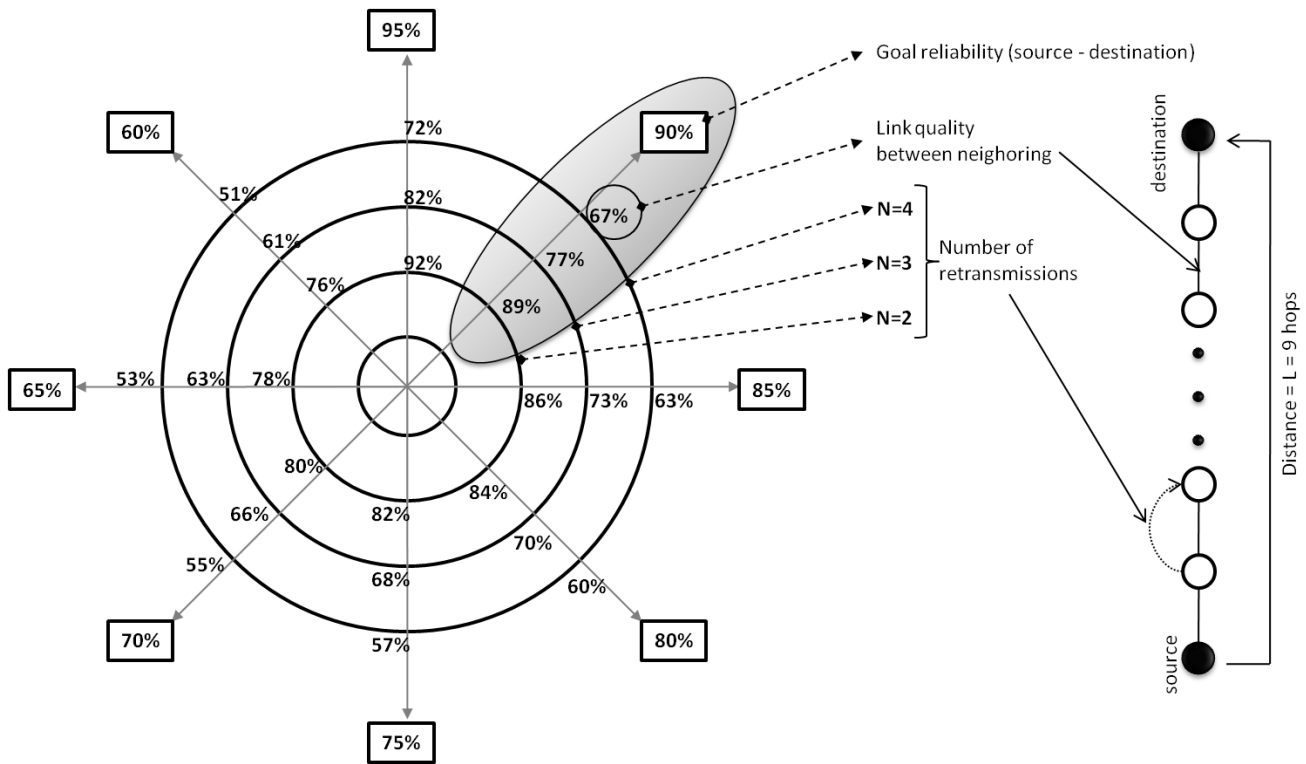


Fig. 2. Reliability analysis between two nodes which are 9 hops away from each other

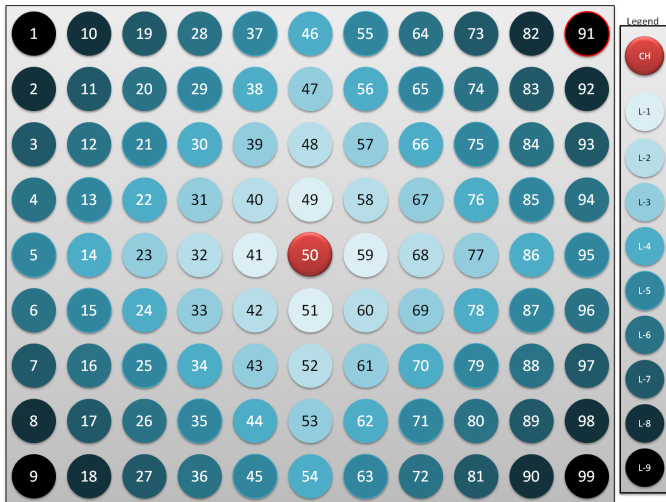


Fig. 3. Evaluation Topology

to achieve a greater reliability in these kinds of situations.

VI. EVALUATION

In order to analyze and study the performance of the reliability model presented in this work, several experiments has been carried out. Figure 3 shows the cluster topology used in the simulations to carry out the experiments. A square grid topology with 99 nodes has been used, as it is quite a standard configuration and in addition, it can also represent very well a cluster of nodes. Basically, the experiments have consisted

in sending 100 packets (events and commands) between the cluster-head (node 50) and a sensor node (node 91) which are 9 hops away from each other, in order to measure how accurate the reliability achieved by the model is.

A. Environment set-up

COOJA sensor network simulator [12] has been used to carry out all the experiments. COOJA is a power profiling tool that enables accurate network-scale energy measurements in a simulated environment. COOJA simulator offers the possibility of carrying out the simulation in different platforms. We selected the TelosB motes since they are one of the most used by the sensornet community. In order to carry out the simulations, we used Contiki [13] which is an open source, highly portable, multi-tasking operating system for memory-efficient networked embedded systems and wireless sensor networks.

B. Results

In order to evaluate our reliable protocol several experiments have been carried out by using the simulator Cooja. The goal of the experiments was to analyze the packet delivery ratio (PDR) of the protocol after sending 100 packets in both directions, from a sensor node to the cluster-head and viceversa. Concretely, the protocol was configured to achieve a reliability level of 80% between two nodes which were 9 hops away from each other. This scenario was studied by using different link quality estimations. Table I shows the results obtained from the simulations:

Packet Delivery Ratio					
Nodes Link Quality	Theoretical Reliability	Reliable Events	No Reliable Events	Reliable Commands	No Reliable Commands
84%	20%	82%	14%	94%	58%
70%	4%	84%	2%	97%	11%
60%	1%	87%	0%	92%	2%
52%	0%	66%	0%	76%	0%

TABLE I
RELIABILITY RESULTS

- Column 1 shows the four different link qualities set in the simulator.
- Column 2 shows the the theoretical reliability between two nodes which are 9 hops away from each other. For example, if the link quality of the nodes is 84%, the reliability between the two nodes mentioned above will be 0.84^9 which is equal to $0.2082 = 20\%$.
- Columns 3 and 4 show the achieved reliability level after sending 100 events (communication from a sensor node to the cluster-head) by using the reliable protocol and the basic protocol.
- Columns 5 and 6 show the achieved reliability level after sending 100 commands (communication from the cluster-head to a sensor node) by using the reliable protocol and the basic protocol.

Let us analyze the results shown in table I. On the one hand, it can be appreciated that the PDR achieved transmitting events is about 80% despite that the theoretical reliability is much more lower. Actually, when the events are sent without using the reliable protocol the results obtained (column four) are similar to the reliability level indicated by the theoretical reliability. On the other hand, the result obtained from the commands are quite different. When they are transmitted the reliability level is higher than 80%. It is due to the fact that when the packets are sent from the cluster-head to the sensor nodes, they can take several paths. Finally, it can also be appreciated that when link qualities of the nodes is set to the 52% the achieved reliability level is lower than 80% (in both cases, event and commands). We believe that it could be due to the collisions produced in the sensor networks since the number of retransmissions is quite high.

VII. CONCLUSIONS

In this paper, we have presented a tool to study and understand how both link quality estimations and distances between source and destination nodes affect the communication reliability. As a novel contribution we have presented a reliability model which allows developers to quantitatively set the desired reliability between a sensor node and its leader node whatever the distance between them is. Finally, a set of experimenters have been carried out to prove the suitability of the proposed model. The results obtained show that when the link quality estimations are greater or equal to 60% the performance of the reliability model is quite accurate.

ACKNOWLEDGMENT

This work was partially supported by Spanish Projects TIN2008-03107 and TIC-03085.

REFERENCES

- [1] I. Akyildiz and M. Vuran, *Wireless sensor networks*. LibreDigital, 2010.
- [2] J. Gehrke and L. Liu, "Sensor-network applications," *IEEE Internet Computing*, vol. 10, no. 2, 2006.
- [3] A. Arora, R. Ramnath, E. Ertin, P. Sinha, S. Bapat, V. Naik, V. Kulathumani, H. Zhang, H. Cao, M. Sridharan *et al.*, "Exscal: Elements of an extreme scale wireless sensor network," 2005.
- [4] A. Willig and H. Karl, "Data transport reliability in wireless sensor networks—a survey of issues and solutions," *Praxis der Informationsverarbeitung und Kommunikation*, vol. 28, no. 2, pp. 86–92, 2005.
- [5] E. C. H. Ngai, Y. Zhou, M. R. Lyu, and J. Liu, "Reliable reporting of delay-sensitive events in wireless sensor-actuator networks," in *Mobile Adhoc and Sensor Systems (MASS), 2006 IEEE International Conference on*, Oct. 2006, pp. 101–108.
- [6] B. Deb, S. Bhatnagar, and B. Nath, "Reinform: reliable information forwarding using multiple paths in sensor networks," in *Local Computer Networks, 2003. LCN '03. Proceedings. 28th Annual IEEE International Conference on*, Oct. 2003, pp. 406–415.
- [7] K. Sharma, H. Singh, and R. Patel, "A Reliable And Energy Efficient Transport Protocol for Wireless Sensor Networks," *Global Journal of Computer Science and Technology*, vol. 10, no. 9, 2010.
- [8] H. Zhou, Y. Wu, Y. Hu, and G. Xie, "A novel stable selection and reliable transmission protocol for clustered heterogeneous wireless sensor networks," *Computer Communications*, 2010.
- [9] J. Paek and R. Govindan, "RCRT: Rate-controlled reliable transport protocol for wireless sensor networks," *ACM Transactions on Sensor Networks (TOSN)*, vol. 7, no. 3, pp. 1–45, 2010.
- [10] E. Felemban, C. Lee, and E. Ekici, "MMSPEED: Multipath multi-SPEED protocol for QoS guarantee of reliability and timeliness in wireless sensor networks," *IEEE Transactions on Mobile Computing*, pp. 738–754, 2006.
- [11] P. Misra, N. Ahmed, D. Ostry, and S. Jha, "Characterization of Asymmetry in Low-Power Wireless Links: An Empirical Study," *Distributed Computing and Networking*, pp. 340–351, 2011.
- [12] J. Eriksson, F. Österlind, N. Finne, A. Dunkels, N. Tsiftes, and T. Voigt, "Accurate Network-Scale Power Profiling for Sensor Network Simulators," *Wireless Sensor Networks*, pp. 312–326, 2009.
- [13] A. Dunkels *et al.*, "Contiki—a lightweight and flexible operating system for tiny networked sensors," 2004.

Fault-tolerant Routing Algorithms based on Approximate Routable Probabilities for Hypercube Networks

Dinh Thuy Duong Keiichi Kaneko
 Department of Computer and Information Sciences
 Graduate School of Engineering
 Tokyo University of Agriculture and Technology
 Koganei-shi, Tokyo, JAPAN

Abstract *Recently, research on parallel processing systems is very active, and many complex topologies have been proposed. One of the most popular topologies is a hypercube network. In this paper, we propose new fault-tolerant routing algorithms for hypercube networks based on approximate routable probabilities. 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 approximate routable 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, the hypercube topology is widely used for interconnection networks for parallel processing due to their properties of regular and recursive structure and low diameter [7]. Figure 1 shows an example of a 4-dimensional hypercube Q_4 . Nodes in a parallel processing system communicate through message passing. Therefore, routing messages is one of the most important problems in parallel processing systems. In addition, the more the number of

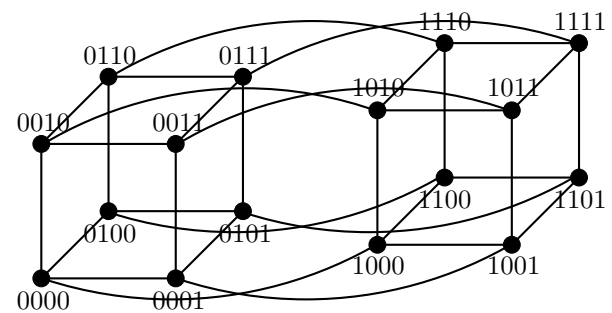


Figure 1: An example of 4-dimensional hypercube Q_4 .

nodes increases, the higher the possibility of occurrence of faulty nodes becomes. Hence, finding a fault-free and shortest path between a source node and a destination node in a parallel processing system with faulty nodes is an emerged problem. A good fault-tolerant routing algorithm must satisfy a couple of conditions. First, the algorithm must find a fault-free path. Second, because of time and space complexities, information stored in each node should be so small that each node cannot identify all of faulty nodes. Therefore, in this paper, we assume that information store in each node must be of polynomial time and space complexities of n in an n -dimensional hypercube Q_n with faulty nodes. With this assumption, we propose two fault-tolerant routing algorithms that find a fault-free path between non-faulty nodes.

The rest of this paper is structured as follows. First, we survey related works in Chapter 2. Next, requisite terminology and notations are defined in Chapter 3. Then, in Chapter 4, we introduce the routing probabilities, their approximate values, and a simplified calculation method for them. Furthermore, we proposed two fault-tolerant routing algorithms in Chapter 5, and evaluate their performance by a computer experiment in Chapter 6. Finally, we give a conclusion and future works in Chapter 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 [3]. 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 [4]. Wu has also proposed a similar fault-tolerant routing algorithm independently by introducing the safety vectors [6]. 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 [5].

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 destina-

tions. However, in the worst case, one non-faulty node has to collect information of all faulty nodes. Hence, it is not possible to finish communication for information exchange in practical time.

In this study, to address the drawback of the method by Al-Sadi et al., we introduce a new concept of approximate routable probabilities and a simplified calculation method for them. Then, we propose two fault-tolerant routing algorithms based on them. Moreover, we carry out a computer experiment to evaluate performance of the algorithms.

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, which consists of 2^n nodes. Each node \mathbf{a} is an n -bit sequence (a_1, a_2, \dots, 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 \mathbf{a} and \mathbf{b} in Q_n , there is an edge (\mathbf{a}, \mathbf{b}) between them if and only if the Hamming distance between them $H(\mathbf{a}, \mathbf{b})$ is equal to 1. \square

In general, a path in a graph is represented by an alternate sequence of nodes and edges $\mathbf{a}_1, (\mathbf{a}_1, \mathbf{a}_2), \mathbf{a}_2, \dots, \mathbf{a}_{k-1}, (\mathbf{a}_{k-1}, \mathbf{a}_k)$. 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 \mathbf{a} and \mathbf{b} is equal to $H(\mathbf{a}, \mathbf{b})$.

Definition 2 For a node \mathbf{a} in Q_n , a set of nodes $N(\mathbf{a})$ defined by

$$N(\mathbf{a}) = \{\mathbf{n} \mid H(\mathbf{a}, \mathbf{n}) = 1\}.$$

is called a set of neighbor nodes of \mathbf{a} . \square

In a hypercube Q_n with a set of faulty nodes F , for a source node \mathbf{s} and a destination node \mathbf{d} that are both non-faulty, a fault-tolerant routing algorithm finds a fault-free path between \mathbf{s} and \mathbf{d} .

Definition 3 In Q_n , for two nodes \mathbf{a} and \mathbf{b} , the set of preferred neighbor nodes of \mathbf{a} for \mathbf{b} is denoted by $N_0(\mathbf{a}, \mathbf{b})$, and is defined by $N_0(\mathbf{a}, \mathbf{b}) = \{\mathbf{n} \mid \mathbf{n} \in N(\mathbf{a}), H(\mathbf{n}, \mathbf{b}) = H(\mathbf{a}, \mathbf{b}) - 1\}$. In addition, the set of spare neighbor nodes of \mathbf{a} for \mathbf{b} is denoted by $N_1(\mathbf{a}, \mathbf{b})$, and is defined by $N_1(\mathbf{a}, \mathbf{b}) = \{\mathbf{n} \mid \mathbf{n} \in N(\mathbf{a}), H(\mathbf{n}, \mathbf{b}) = H(\mathbf{a}, \mathbf{b}) + 1\}$. \square

Note that, in Q_n , the number of nodes that are apart from a node \mathbf{a} by Hamming distance h is equal to ${}_n C_h$. Note also that, for two nodes \mathbf{a} and \mathbf{b} in Q_n , if $H(\mathbf{a}, \mathbf{b}) = h$, then $|N_0(\mathbf{a}, \mathbf{b})| = h$ holds.

4 Routable Probabilities

In this section, we give the idea of routable probabilities. In an n -dimensional hypercube Q_n with a set of faulty nodes F , the routable probability $P_h^*(\mathbf{a})$ of a non-faulty node \mathbf{a} with respect to a Hamming distance h represents the probability that, for an arbitrary non-faulty node \mathbf{b} with $h = H(\mathbf{a}, \mathbf{b})$, there is a fault-free path of length h between \mathbf{a} and \mathbf{b} .

Since it is difficult to calculate the routable probabilities precisely, we use the following approximate values.

Definition 4 For a node \mathbf{a} in an n -dimensional hypercube Q_n with a set of faulty nodes F , approximate probabilities $P_h(\mathbf{a})$ of \mathbf{a} with respect to Hamming distance h is defined as follows:

$$P_h(\mathbf{a}) = \begin{cases} 1 & (h = 0) \\ 0 & (1 \leq h \leq n, \mathbf{a} \in F) \\ \sum_{\substack{I \subset N(\mathbf{a}) \\ |I|=h}} \max_{\mathbf{n} \in I} \{P_{h-1}(\mathbf{n})\} / {}_n C_h & (1 \leq h \leq n, \mathbf{a} \notin F) \end{cases}$$

\square

Definition 4 for $P_h(\mathbf{a})$ has the following meaning. First, an arbitrary node including a faulty node can send a message to itself with probability 1. Next, if a node \mathbf{a} is faulty, it cannot send a message to any node other than

itself. Hence, for any positive Hamming distance h , $P_h(\mathbf{a}) = 0$ holds. Otherwise, when we take h nodes arbitrarily from the set of neighbor nodes $N(\mathbf{a})$ of \mathbf{a} , the expectation value of the maximum routable probabilities of these nodes with respect to Hamming distance $h - 1$ is calculated.

The approximate routable probabilities with respect to Hamming distance 0 is defined to be 1 for all the nodes including faulty nodes. Therefore, for any non-faulty node \mathbf{a} , $P_1(\mathbf{a}) = 1$ holds.

To calculate the approximate routable probabilities easily, we introduce the following theorem.

Theorem 1 In an n -dimensional hypercube Q_n with a set of faulty nodes F , for a non-faulty node \mathbf{a} and a natural number h ($1 \leq h \leq n$),

$$P_h(\mathbf{a}) = \sum_{k=h}^n {}_{k-1} C_{h-1} p_k / {}_n C_h$$

where $p_1 \leq p_2 \leq \dots \leq p_n$ are obtained by sorting $P_{h-1}(\mathbf{a} \oplus 2^0)$, $P_{h-1}(\mathbf{a} \oplus 2^1)$, \dots , $P_{h-1}(\mathbf{a} \oplus 2^{n-1})$.

(Proof) In Definition 4, $p_k = \max_{\mathbf{n} \in I} \{P_{h-1}(\mathbf{n})\}$ holds if and only if $p_k \in \cup_{\mathbf{n} \in I} \{P_{h-1}(\mathbf{n})\}$ and $\cup_{\mathbf{n} \in I} \{P_{h-1}(\mathbf{n})\} \subset \{p_1, p_2, \dots, p_k\}$ hold. Therefore, the number of occurrences such that p_k becomes the maximum value is equal to ${}_{k-1} C_{h-1}$. Hence, the theorem holds. \square

From Theorem 1, we can obtain a function ARP to calculate the approximate routable probabilities. It is shown in Figure 2.

In the rest of this paper, for a non-faulty node \mathbf{a} in an n -dimensional hypercube Q_n , if $N(\mathbf{a}) \subset F$ holds, then we take \mathbf{a} as faulty. Then, the following theorem holds.

Theorem 2 In an n -dimensional hypercube Q_n with a set of faulty nodes F , for a node \mathbf{a} , there is a natural number h ($1 \leq h \leq n$) such that $P_h(\mathbf{a}) = 0$ if and only if $\mathbf{a} \in F$ holds. (Proof) If $\mathbf{a} \in F$, then for any h such that

```

function ARP( $\mathbf{a}$ ,  $h$ ,  $F$ )
begin
  if  $h = 0$  then  $P_h(\mathbf{a}) := 1$ 
  else if  $\mathbf{a} \in F$  then  $P_h(\mathbf{a}) := 0$ 
  else begin
    collect  $\{P_{h-1}(\mathbf{n}) \mid \mathbf{n} \in N(\mathbf{a})\}$ ;
    sort  $\{P_{h-1}(\mathbf{n}) \mid \mathbf{n} \in N(\mathbf{a})\}$ 
    to obtain  $p_1 \leq p_2 \leq \dots \leq p_n$ ;
     $P_h(\mathbf{a}) := \sum_{k=1}^n C_{h-1} p_k / C_h$ 
  end;
  return  $P_h(\mathbf{a})$ 
end

```

Figure 2: Function to calculate approximate values of routable probabilities

$1 \leq h \leq n$, $P_h(\mathbf{a}) = 0$ holds from Definition 4. For a natural number h ($1 \leq h \leq n$), we prove $\mathbf{a} \in F$ assuming $P_h(\mathbf{a}) = 0$ by induction on h . Assume $P_1(\mathbf{a}) = 0$. If $\mathbf{a} \notin F$, $P_1(\mathbf{a}) = \sum_{\mathbf{n} \in N(\mathbf{a})} P_0(\mathbf{n})/n = 1$ holds from Definition 4. Because of $P_1(\mathbf{a}) = 0$, $\mathbf{a} \in F$ holds. If $P_h(\mathbf{a}) = 0$ for h such that $2 \leq h \leq n$, then, from the definition of $P_h(\mathbf{a})$, $P_{h-1}(\mathbf{n}) = 0$ holds for an arbitrary neighbor node \mathbf{n} of \mathbf{a} . From the hypothesis of induction, $N(\mathbf{a}) \subset F$ holds. From this, $\mathbf{a} \in F$ holds. \square

From Theorem 2, if a node \mathbf{a} is non-faulty, then $P_h(\mathbf{a}) > 0$ holds for an arbitrary h such that $0 \leq h \leq n$.

5 Fault-tolerant Routing Algorithms

In this section, we show how to find a path in a hypercube with faulty nodes by using approximate routable probabilities. The routing strategy is based on the approximate routable probabilities of neighbor nodes stored in each node.

In an n -dimensional hypercube with a set of faulty nodes F , we assume that each node \mathbf{a} stores the approximate routable probabilities for all of the neighbor nodes \mathbf{n} ($\in N(\mathbf{a})$) with respect to all Hamming distances h ($0 \leq h \leq$

n). Then, for a non-faulty source node \mathbf{s} and a non-faulty destination node \mathbf{d} , we propose two fault-tolerant routing algorithms which establish fault-free paths between them.

5.1 Naive Algorithm DK0

First, we propose a simple fault-tolerant routing algorithm DK0. It takes the current node \mathbf{c} and the destination node \mathbf{d} as its arguments. Then, Algorithm DK0 selects the node among the preferred neighbor nodes of the current node for the destination node that has the largest positive approximate routable probability with respect to $H(\mathbf{c}, \mathbf{d}) - 1$, and send the message to the selected neighbor node. If the approximate routable probabilities of the preferred nodes are all zero, then the node with the largest approximate probability with respect to $H(\mathbf{c}, \mathbf{d}) + 1$ is selected among the spare neighbor nodes, and the message is sent to it.

Figure 3 shows the pseudo code for the algorithm where exception handling for the case $h = n$ is omitted. From Theorem 2, $P_{h-1}(\mathbf{n}_0^*) > 0$ or $P_{h+1}(\mathbf{n}_1^*) > 0$ holds. Hence, the routing always fails by an infinite loop.

```

procedure DK0( $\mathbf{c}$ ,  $\mathbf{d}$ )
begin
   $h := H(\mathbf{c}, \mathbf{d})$ ;
   $\mathbf{n}_0^* := \arg \max_{\mathbf{n} \in N_0(\mathbf{c}, \mathbf{d})} \{P_{h-1}(\mathbf{n})\}$ ;
   $\mathbf{n}_1^* := \arg \max_{\mathbf{n} \in N_1(\mathbf{c}, \mathbf{d})} \{P_{h+1}(\mathbf{n})\}$ ;
  if  $h = 0$  then
    deliver the message to  $\mathbf{c}$ 
  else if  $P_{h-1}(\mathbf{n}_0^*) > 0$  then DK0( $\mathbf{n}_0^*$ ,  $\mathbf{d}$ )
  else DK0( $\mathbf{n}_1^*$ ,  $\mathbf{d}$ )
end

```

Figure 3: Routing algorithm DK0

5.2 Improved Algorithm DK1

Next, we propose an alternate fault-tolerant routing algorithm DK1. It takes the previous node \mathbf{p} , the current node \mathbf{c} , and the destination node \mathbf{d} . From the previous node, the message is sent to the current node. Algorithm DK1 selects the node among the preferred neighbor

nodes $N_0(\mathbf{c}, \mathbf{d})$ of the current node for the destination node that has the largest positive approximate routable probability with respect to $H(\mathbf{c}, \mathbf{d}) - 1$, and send the message to the selected neighbor node. This is same as Algorithm DK0. If the approximate routable probabilities of the preferred nodes are all zero, then the node with the largest approximate probability with respect to $H(\mathbf{c}, \mathbf{d}) + 1$ is selected among the spare neighbor nodes except for the previous node $N_1(\mathbf{c}, \mathbf{d}) \setminus \{\mathbf{p}\}$, and the message is sent to it. Note that Algorithm DK1 excludes the previous node \mathbf{p} from the candidate nodes of the spare nodes. This exclusion avoids a simple loop of message sending between two nodes.

Figure 4 shows the pseudo code for the algorithm where exception handling for the case $k = n$ is omitted. Note that Algorithm dk1 may explicitly fail to send a message.

```

procedure DK1( $\mathbf{p}, \mathbf{c}, \mathbf{d}$ )
begin
   $h := H(\mathbf{c}, \mathbf{d});$ 
   $\mathbf{n}_0^* := \arg \max_{\mathbf{n} \in N_0(\mathbf{c}, \mathbf{d})} \{P_{h-1}(\mathbf{n})\};$ 
   $\mathbf{n}_1^* := \arg \max_{\mathbf{n} \in N_1(\mathbf{c}, \mathbf{d}) \setminus \{\mathbf{p}\}} \{P_{h+1}(\mathbf{n})\};$ 
  if  $h = 0$  then
    deliver the message to  $\mathbf{c}$ 
  else if  $P_{h-1}(\mathbf{n}_0^*) > 0$  then
    DK1( $\mathbf{c}, \mathbf{n}_0^*, \mathbf{d}$ )
  else if  $P_{h+1}(\mathbf{n}_1^*) > 0$  then
    DK1( $\mathbf{c}, \mathbf{n}_1^*, \mathbf{d}$ )
  else error('unable to deliver')
end

```

Figure 4: Routing algorithm DK1

6 Performance Evaluation

In this section, we first analyze the time complexity of calculation of approximate routable probabilities, which is the first step of our algorithms. Next, we compare our algorithms and the algorithm by Al-Sadi et al. [1, 2] by a computer experiment.

6.1 Time Complexity

Time complexity for calculation of approximate routable probabilities in each node depends on the expression $P_h(\mathbf{a}) = \sum_{k=1}^n {}^{k-1}C_{h-1} p_k / {}_n C_h$. Combinations of ${}^{k-1}C_{h-1}$ and ${}_n C_h$ are calculated at first, and stored in an array. It takes $O(n^3)$ time complexity. For a Hamming distance h , sorting of p_k ($1 \leq k \leq n$) takes $O(n \log n)$ time complexity, and calculation of $P_h(\mathbf{a})$ takes $O(n)$ time complexity. Therefore, for all h ($1 \leq h \leq n$), sorting p_k 's and calculation of $P_h(\mathbf{a})$ require $O(n^2 \log n)$ time complexity. From the above discussion, calculation of the table of combinations is dominant, and it takes $O(n^3)$ time complexity in total. Each node has to exchange information n times with each of its neighbor nodes.

6.2 Computer Experiment

In this section, we give the detail of the results of a computer experiment conducted to compare Algorithms DK0 and DK1 we proposed, and Algorithms AD00 and AD01 by Al-Sadi et al. Algorithm AD00 is the original algorithm proposed by Al-Sadi et al. while Algorithm AD01 is obtained by restricting the spare nodes as in Algorithm DK1 to suppress infinite loops between two nodes.

A computer experiment was carried out for n -dimensional hypercubes where $5 \leq n \leq 10$ changing the ratio of faulty nodes ρ from 10% to 80%. Concretely, first, in Q_n ($5 \leq n \leq 10$), we selected faulty nodes randomly with the ratio ρ . Next, we selected the source node \mathbf{s} and the destination node \mathbf{d} from non-faulty nodes. Finally, after checking the connectivity of \mathbf{s} and \mathbf{d} , we applied the four fault-tolerant routing algorithms to measure the ratio of successful routings. If \mathbf{s} and \mathbf{d} are not connected, that is, there is no fault-free path between them, we start over from the selection of faulty nodes. For each pair of a dimension and a ratio of faulty nodes, we executed at least 1,000 trials.

Figures 5, 6, 7, 8 show the results by Algorithms DK0, DK1, AD00 and AD01, respectively.

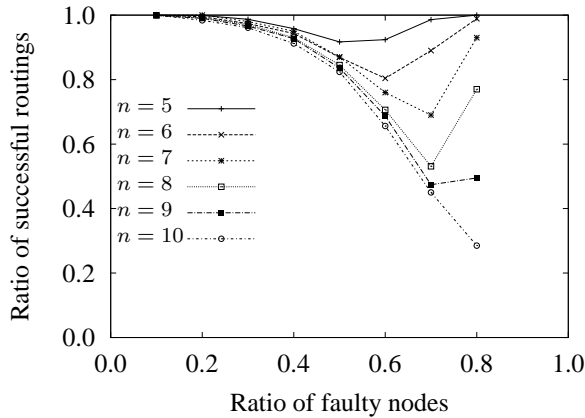


Figure 5: Ratio of successful routings by Algorithm DK0

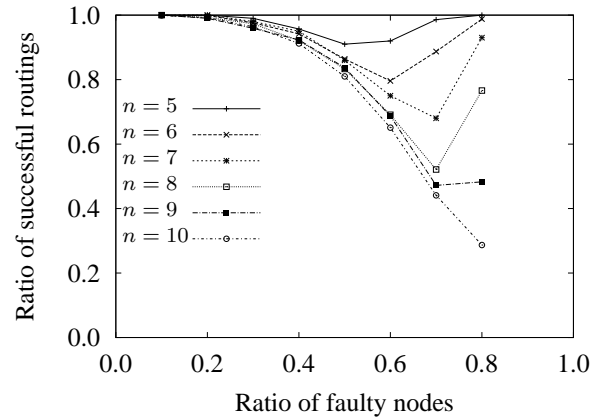


Figure 7: Ratio of successful routings by Algorithm AD00

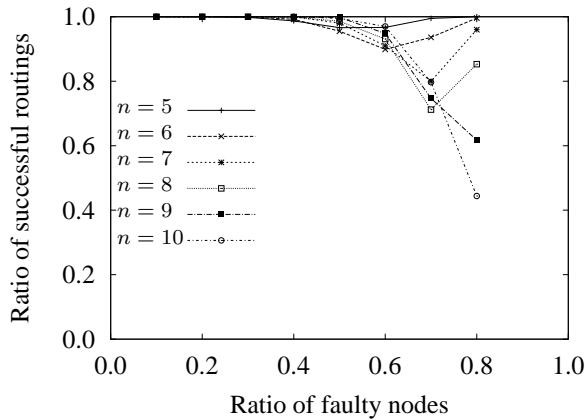


Figure 6: Ratio of successful routings by Algorithm DK1

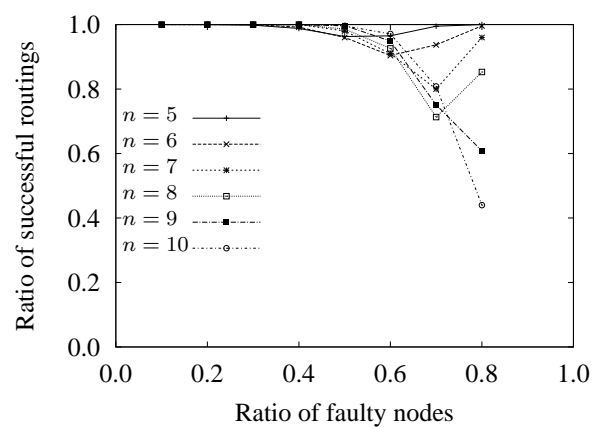


Figure 8: Ratio of successful routings by Algorithm AD01

As a result, we can see that performance of DK0 and DK1 is almost equivalent to that of AD00 and AD01, respectively.

From the discussion above, we can conclude that the proposed algorithms have equivalent routing ability to the algorithms by Al-Sadi with the much lower time complexity.

7 Conclusion

In this paper, we have proposed two new fault-tolerant routing algorithms for hypercube networks based on approximate routable probabilities, which represent 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 approximate routable probabilities into consideration.

We also conducted a computer experiment to verify the effectiveness of our algorithms. As a result, we proved that our algorithms have almost equivalent performance to the algorithms proposed by Al-Sadi et al.

Our next step is to extend the concept of the approximate routable probabilities to apply other topologies for parallel processing systems. The path lengths are also an important problem. Therefore, we also intend to improve the routing algorithm so it can solve out the shorter fault-free paths between any fault-free

nodes in a hypercube.

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*, 44(5), 2001.
- [2] J. Al-Sadi, K. Day, and M. Ould-Khaoua. Fault-tolerant routing in hypercubes using probability vectors. *Parallel Computing*, 27, 1381–1399, 2001.
- [3] G.-M. Chiu and S.-P. Wu. A fault-tolerant routing strategy in hypercube multicomputers. *IEEE Transactions on Computers*, 45(2):143-155, Feb 1996.
- [4] G.-M. Chiu and K.-S. Chen. Use of routing capability for fault-tolerant routing in hypercube multicomputers. *IEEE Transactions on Computers*, 46(8), 1997.
- [5] K. Kaneko and H. Ito. Fault-tolerant routing algorithms for hypercube interconnection networks. *IEICE Transactions on Information and Systems*, E84-D(1):121–128, Jan 2001.
- [6] 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.
- [7] C. L. Seitz. The cosmic cube. *Communications of ACM*, 28(7):22–33, Jul 1985.

The Hyper-Panconnectedness of the Crossed Cube*

Hon-Chan Chen¹, Tzu-Liang Kung², and Lih-Hsing Hsu³

¹Department of Information Management, National Chin-Yi University of Technology, Taichung, Taiwan

²Department of Computer Science and Information Engineering, Asia University, Taichung, Taiwan

³Department of Computer Science and Information Engineering, Providence University, Taichung, Taiwan

Abstract - A connected graph G is said to be panconnected if, for any two distinct vertices x and y of G , it contains a path P between x and y such that $d_P(x, y) = l$ for any integer l satisfying $d_G(x, y) \leq l \leq |V(G)| - 1$, where $d_G(x, y)$ (respectively, $d_P(x, y)$) denotes the distance between x and y in G (respectively, P), and $|V(G)|$ denotes the total number of vertices of G . If such path P can be extended to form a Hamiltonian path P' of G such that $P \subseteq P'$, then G is hyper-panconnected. In this paper, we study the property of hyper-panconnectedness with respect to the class of crossed cubes, which is a popular variant of the hypercube network.

Keywords: Panconnected, Hamiltonian path, Crossed cube, Interconnection network

1 Introduction

Network topology is essential for parallel and distributed computation, and it determines the performance of a network. Among many kinds of network topologies, the hypercube [14] is one of the most popular networks since it has good properties such as regularity, symmetry, small diameter, strong connectivity, recursive construction, partitionability, relatively low link complexity, and so on. However, the hypercube is bipartite so that it cannot make the best use of network resources. To compensate for this drawback, many researchers [1, 5, 6, 15] try to fashion networks. One such network topology is the crossed cube, which was first proposed by Efe [7]. The crossed cube is derived from the hypercube by changing the connection of some links. Its diameter is about half of the hypercube's

[4, 7]. Moreover, the crossed cube has additional attractive properties. For example, it has more cycles than the hypercube [9], and it can embed binary trees [12], paths of odd and even lengths [8, 10], and many-to-many disjoint path covers [13]. The definition of the crossed cube will be presented in the next section.

In terms of network analysis, the topological structure can be modeled as a simple graph. For the graph definition and notation, we follow the standard terminology given by Bondy and Murty [3]. Let G be an undirected graph with vertex set $V(G)$ and edge set $E(G)$. Two vertices x and y of G are *adjacent* if $(x, y) \in E(G)$. The *degree* of a vertex u is the number of edges incident to u . A graph G is *k-regular* if all its vertices have the same degree k . A graph H is a *subgraph* of G , denoted by $H \subseteq G$, if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. Moreover, H is a *spanning subgraph* of G (or H *spans* G) if $V(H) = V(G)$. Let S be a nonempty subset of $V(G)$. The subgraph of G *induced* by S is a graph whose vertex set is S and whose edge set consists of all the edges of G joining any two vertices in S . A *path* P of length k , $k \geq 1$, from vertex x to vertex y in G is a sequence of distinct vertices $\langle v_1, v_2, \dots, v_{k+1} \rangle$ such that $v_1 = x$, $v_{k+1} = y$, and $(v_i, v_{i+1}) \in E(G)$ for $1 \leq i \leq k$. We can write P as $\langle v_1, v_2, \dots, v_i, Q, v_j, \dots, v_{k+1} \rangle$ for convenience if we know that $Q = \langle v_i, \dots, v_j \rangle$, where $i \leq j$. The *reverse* of P , denoted by $rev(P)$, is defined as $rev(P) = \langle v_{k+1}, v_k, \dots, v_1 \rangle$. A single vertex x can be considered as a path $\langle x \rangle$ of length 0. The i th vertex of P is denoted by $P(i)$; i.e., $P(i) = v_i$. We use $l(P)$ to denote the *length* of P . The *distance* between two distinct vertices x and y in G , denoted by $d_G(x, y)$, is the length of the shortest path between x and y . A *cycle* is a closed path with at least three vertices such that the last vertex is adjacent to the first one. For clarity, a cycle of length k is represented by $C_k = \langle v_1, v_2, \dots, v_k, v_1 \rangle$, where $k \geq 3$.

* This work is supported in part by the National Science Council of the Republic of China under Contracts NSC98-2218-E-468-001-MY3 and NSC99-2221-E-167-025.

A path (respectively, cycle) is a *Hamiltonian path* (respectively, *Hamiltonian cycle*) of G if it spans G . A graph G is *Hamiltonian* if it has a Hamiltonian cycle, and a graph G is *Hamiltonian connected* if it contains a Hamiltonian path between any two distinct vertices. A graph G is said to be *panconnected* [2] if, for any two distinct vertices x and y , it has a path of length l joining x and y for any integer l satisfying $d_G(x,y) \leq l \leq |V(G)| - 1$. From the above definitions, we know that for any two distinct vertices x and y of a panconnected graph G , there exists a path P of required length l joining x and y in G , but we do not know whether such P can be extended to form a Hamiltonian path of G when $l < |V(G)| - 1$. Therefore, we define the property of *hyper-panconnectedness* as follows: for any two distinct vertices x and y of G , if there exists a Hamiltonian path P of G such that $d_P(x,y) = l$ for any integer l satisfying $d_G(x,y) \leq l \leq |V(G)| - 1$, then G is *hyper-panconnected*.

In this paper, we study the hyper-panconnectedness for the crossed cube. The rest of this paper is organized as follows. In Section 2, the definition and some properties of the crossed cube are introduced. In Section 3, we propose our main theorem and show its correctness. Finally, some concluding remarks are given in Section 4.

2 The crossed cube

A crossed cube of n -dimensions, denoted by CQ_n , has 2^n vertices. Each vertex of CQ_n is identified by a unique n -bit binary string; e.g. vertex $u = u_n u_{n-1} \dots u_2 u_1$, where $u_i \in \{0, 1\}$ for $1 \leq i \leq n$. The following are the formal definitions about the crossed cube [6].

Definition 1. Two binary strings $x = x_2 x_1$ and $y = y_2 y_1$ of length two are pair related, denoted by $x \sim y$, if and only if $(x,y) \in \{(00,00), (10,10), (01,11), (11,01)\}$.

Definition 2. The n -dimensional crossed cube CQ_n is recursively constructed as follows. CQ_1 is a complete graph with two vertices 0 and 1. CQ_n , $n \geq 2$, consists of two identical $(n-1)$ -dimensional crossed cubes CQ_{n-1}^0 and CQ_{n-1}^1 , and a vertex $u = 0u_{n-1} \dots u_2 u_1 \in V(CQ_{n-1}^0)$ is adjacent to a vertex $v = 1v_{n-1} \dots v_2 v_1 \in V(CQ_{n-1}^1)$ in CQ_n if and only if

- (1) $u_{n-1} = v_{n-1}$ if n is even, and

- (2) $u_{2i} u_{2i-1} \sim v_{2i} v_{2i-1}$ for all i , $1 \leq i \leq \lfloor \frac{n-1}{2} \rfloor$.

From the above definition, CQ_2 is just a C_4 , and CQ_n is an n -regular graph. Figure 1 shows CQ_3 and CQ_4 .

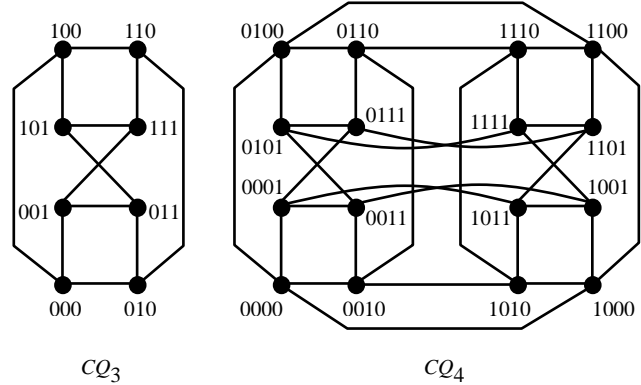


Figure 1: Illustration of CQ_3 and CQ_4 .

In [7], Efe proposed a shortest path routing algorithm $Route(x,y)$ for CQ_n . This algorithm implies the following two lemmas.

Lemma 1. [7] Let x and y be any two distinct vertices of CQ_n such that x and y are in $V(CQ_{n-1}^i)$, $i \in \{0,1\}$. Then, $d_{CQ_n}(x,y) = d_{CQ_{n-1}^i}(x,y)$.

Lemma 2. [7] Let x and y be any two vertices of CQ_n such that $x \in V(CQ_{n-1}^0)$ and $y \in V(CQ_{n-1}^1)$. Moreover, let a be the vertex in $V(CQ_{n-1}^1)$ adjacent to x , and let b be the vertex in $V(CQ_{n-1}^0)$ adjacent to y . Then, $d_{CQ_n}(x,b) = d_{CQ_n}(x,y) - 1$ or $d_{CQ_n}(a,y) = d_{CQ_n}(x,y) - 1$.

Fan et al. [10] have proved that paths of various lengths can be embedded into CQ_n .

Lemma 3. [10] Let x and y be any two distinct vertices of CQ_n . Moreover, let l be any integer with $d_{CQ_n}(x,y) \leq l \leq 2^n - 1$ and $l \neq d_{CQ_n}(x,y) + 1$. Then, there exists a path of length l joining x and y in CQ_n .

A Hamiltonian graph G is said to be *f-fault-tolerant Hamiltonian* if $G - F$ remains Hamiltonian for every $F \subseteq V(G) \cup E(G)$ with $|F| \leq f$. A Hamiltonian connected graph G is said to be *f-fault-tolerant Hamiltonian connected* if $G - F$ remains Hamiltonian connected for every $F \subseteq V(G) \cup E(G)$ with $|F| \leq f$.

Lemma 4. [11] CQ_n is $(n-2)$ -fault-tolerant Hamiltonian and $(n-3)$ -fault-tolerant Hamiltonian connected for any integer $n, n \geq 3$.

In CQ_n , vertex $x = x_n x_{n-1} \dots x_1$ is adjacent to vertex $y = y_n y_{n-1} \dots y_1$ along the i th dimension, $1 \leq i \leq n$, if the following four conditions are all satisfied: (i) $x_i \neq y_i$, (ii) $x_j = y_j$ for all $j, i+1 \leq j \leq n$, (iii) $x_{2k} x_{2k-1} \sim y_{2k} y_{2k-1}$ for all $k, 1 \leq k \leq \lfloor \frac{i-1}{2} \rfloor$, and (iv) $x_{i-1} = y_{i-1}$ if i is even. Then, we say that x is the i -neighbor of y , and vice versa. The i -neighbor of vertex x is denoted by $(x)^i$, and edge $(x, (x)^i)$ is called the i -dimensional edge. It is easy to see that $y = (x)^i$ if and only if $x = (y)^i$. According to the adjacency of vertices, we can locate C_4 and C_5 in CQ_n as described in the following lemmas.

Lemma 5. [9] Let (x, y) be any n -dimensional edge in $CQ_n, n \geq 3$. Then, the set of vertices $\{x, y, (x)^2, (y)^2\}$ induces a C_4 .

Lemma 6. [9] Let (x, y) be any n -dimensional edge in $CQ_n, n \geq 3$. Then, $((x)^1)^n = ((y)^2)^1 = ((y)^1)^2$. Moreover, (i) the set of vertices $\{x, y, (x)^1, (y)^2, ((y)^2)^1\}$ induces a C_5 ; (ii) the set of vertices $\{x, y, (x)^1, (y)^1, ((y)^1)^2\}$ induces a C_5 .

By brute force with a computer program, we have the following lemma for CQ_4 .

Lemma 7. Let (x, y) be any 2-dimensional, 3-dimensional, or 4-dimensional edge of CQ_4 . Then, $CQ_4 - \{x, y\}$ has a Hamiltonian path between two arbitrary vertices.

Corollary 1 is drawn from Lemmas 4 and 7.

Corollary 1. Let (x, y) be any i -dimensional edge of CQ_n , where $n \geq 4$ and $2 \leq i \leq n$. Then, $CQ_n - \{x, y\}$ has a Hamiltonian path between two arbitrary vertices.

3 The hyper-panconnectedness of CQ_n

In Section 1, we define that a graph G is hyper-panconnected if, for any two distinct vertices x and y , there exists a Hamiltonian path P of G such that $d_P(x, y) = l$ for any integer l satisfying $d_G(x, y) \leq l \leq |V(G)| - 1$. For convenience, we can consider $P(1) = x$ and $P(l+1) = y$. However, by the definition of the crossed cube, CQ_n has no C_3 as a subgraph; thus, there does not exist any path of

length 2 between any adjacent vertices x and y in CQ_n . For this reason and by Lemma 3, we exclude the cases of paths of length $d_{CQ_n}(x, y) + 1$ for any two distinct vertices x and y , and we present a loose version of the hyper-panconnectedness for the crossed cube.

Theorem 1. Let x and y be any two distinct vertices of $CQ_n, n \geq 4$. Moreover, let l be any integer with $d_{CQ_n}(x, y) \leq l \leq 2^n - 1$ and $l \neq d_{CQ_n}(x, y) + 1$. Then, there exists a Hamiltonian path P of CQ_n such that $P(1) = x$ and $P(l+1) = y$.

Proof. This theorem will be shown by induction on n . For the induction base CQ_4 , the correctness can be verified by brute force with a computer program. Suppose that this theorem holds for any $CQ_k, 4 \leq k \leq n-1$. Then, we will show that CQ_n has a Hamiltonian path P such that $P(1) = x$ and $P(l+1) = y$. Without loss of generality, assume that $x \in V(CQ_{n-1}^0)$. Consider the following three cases.

Case 1. $y \in V(CQ_{n-1}^0)$. By Lemma 1, we have $d_{CQ_n}(x, y) = d_{CQ_{n-1}^0}(x, y)$. The following three subcases have to be considered.

Subcase 1.1. $2^{n-1} + 1 \leq l \leq 2^n - 1$. By Lemma 4, there exists a Hamiltonian path R of CQ_{n-1}^0 joining x to y . We can write path R as $\langle x, R_1, a, b, R_2, y \rangle$, where a and b are two adjacent vertices in CQ_{n-1}^0 satisfying $l(R_1) = l - 2^{n-1} - 1$. Notice that $x = a$ if $l = 2^{n-1} + 1$ and $b = y$ if $l = 2^n - 1$. Also by Lemma 4, CQ_{n-1}^1 has a Hamiltonian path S joining $(a)^n$ to $(y)^n$. Then, $P = \langle x, R_1, a, (a)^n, S, (y)^n, y, rev(R_2), b \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(l+1) = y$.

Subcase 1.2. $l = 2^{n-1}$. Let a be any vertex in CQ_{n-1}^1 other than $(x)^n$ and $(y)^n$. By Lemma 4, there exists a Hamiltonian path S of $CQ_{n-1}^1 - \{a\}$ joining $(x)^n$ to $(y)^n$. Similarly, there exists a Hamiltonian path R of $CQ_{n-1}^0 - \{x\}$ joining $(a)^n$ and y . Then, $P = \langle x, (x)^n, S, (y)^n, y, R, (a)^n, a \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(2^{n-1} + 1) = y$.

Subcase 1.3. $d_{CQ_n}(x, y) \leq l \leq 2^{n-1} - 1$ and $l \neq d_{CQ_n}(x, y) + 1$. By the inductive hypothesis, there exists a Hamiltonian path R of CQ_{n-1}^0 such that $R(1) = x$ and $R(l+1) = y$. We can write path R as $\langle x, R_1, y, R_2, a \rangle$, where a is some vertex of CQ_{n-1}^0 . Notice that $a = y$ if $l = 2^{n-1} - 1$. By lemma 4, there exists a Hamiltonian path S of CQ_{n-1}^1 joining $(a)^n$ to

some vertex b of CQ_{n-1}^1 , and $P = \langle x, R_1, y, R_2, a, (a)^n, S, b \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(l+1) = y$.

Case 2. $y \in V(CQ_{n-1}^1)$ and $(x, y) \notin E(CQ_n)$. The following subcases are distinguished.

Subcase 2.1. $l = 2^n - 1$. Trivially, there exists a Hamiltonian path P of CQ_n joining x to y .

Subcase 2.2. $2^{n-1} \leq l \leq 2^n - 2$. Let $h = l - 2^{n-1}$. By Lemma 4, CQ_{n-1}^0 has a Hamiltonian path R joining x and $(y)^n$. We can write path R as $\langle x, R_1, a, b, R_2, (y)^n \rangle$, where a and b are two adjacent vertices in CQ_{n-1}^0 with $l(R_1) = h$. Notice that $a = x$ if $l = 2^{n-1}$ and $b = (y)^n$ if $l = 2^n - 2$. By Lemma 4, CQ_{n-1}^1 has a Hamiltonian path S joining $(a)^n$ to y , and $P = \langle x, R_1, a, (a)^n, S, y, (y)^n, rev(R_2), b \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(l+1) = y$.

Subcase 2.3. $d_{CQ_n}(x, y) \leq l \leq 2^{n-1} - 1$ and $l \neq d_{CQ_n}(x, y) + 1$. By Lemma 2, we have two conditions: $d_{CQ_{n-1}^0}(x, (y)^n) = d_{CQ_n}(x, y) - 1$ or $d_{CQ_{n-1}^1}((x)^n, y) = d_{CQ_n}(x, y) - 1$. Firstly, we assume that $d_{CQ_{n-1}^0}(x, (y)^n) = d_{CQ_n}(x, y) - 1$. By the inductive hypothesis, there exists a Hamiltonian path R of CQ_{n-1}^0 with $R(1) = x$ and $R(l) = (y)^n$. For clarity, path R is written as $\langle x, R_1, (y)^n, a, R_2, b \rangle$, where a is a vertex of CQ_{n-1}^0 adjacent to $(y)^n$ and b is some vertex of CQ_{n-1}^0 . Notice that $a = b$ if $l = 2^{n-1} - 1$. By Lemma 4, there exists a Hamiltonian path S of CQ_{n-1}^1 joining y to $(b)^n$, and $P = \langle x, R_1, (y)^n, y, S, (b)^n, b, rev(R_2), a \rangle$ is a Hamiltonian path of CQ_n such that $P(1) = x$ and $P(l+1) = y$. Next, we assume that $d_{CQ_{n-1}^1}((x)^n, y) = d_{CQ_n}(x, y) - 1$. By the inductive hypothesis, there exists a Hamiltonian path S of CQ_{n-1}^1 with $S(1) = (x)^n$ and $S(l) = y$. The path S can be written as $\langle (x)^n, S_1, y, S_2, a \rangle$, where a is some vertex of CQ_{n-1}^1 . By Lemma 4, there exists a Hamiltonian path R of $CQ_{n-1}^0 - \{x\}$ joining $(a)^n$ to some vertex b of CQ_{n-1}^0 . Then, $P = \langle x, (x)^n, S_1, y, S_2, a, (a)^n, R, b \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(l+1) = y$.

Case 3. $y \in V(CQ_{n-1}^1)$ and $(x, y) \in E(CQ_n)$. Since $(x, y) \in E(CQ_n)$, we have $1 \leq l \leq 2^n - 1$ and $l \neq 2$. Consider the following subcases.

Subcase 3.1. $2^{n-1} + 3 \leq l \leq 2^n - 1$. Let $h = l - 2^{n-1}$, and we have $3 \leq h \leq 2^{n-1} - 1$. By Lemma 5, vertices $x, y, (x)^2$, and $(y)^2$ induce a C_4 . By Lemma 4, there exists a

Hamiltonian path R of CQ_{n-1}^0 joining x and $(x)^2$. By the inductive hypothesis, there exists a Hamiltonian path S of CQ_{n-1}^1 with $S(1) = (y)^2$ and $S(h+1) = y$. We can write S as $\langle (y)^2, S_1, y, S_2, a \rangle$, where a is some vertex of CQ_{n-1}^1 . Notice that $a = y$ if $h = 2^{n-1} - 1$. Then, $P = \langle x, R, (x)^2, (y)^2, S_1, y, S_2, a \rangle$ is a Hamiltonian path of CQ_n such that $P(1) = x$ and $P(l+1) = y$.

Subcase 3.2. $l = 2^{n-1} + 2$. By Lemma 6, vertices $x, y, (x)^1, (y)^1$, and $((y)^1)^2$ induce a C_5 . By Lemma 4, there exists a Hamiltonian path R of CQ_{n-1}^0 joining x to $(x)^1$. By Corollary 1, there exists a Hamiltonian path S of $CQ_{n-1}^1 - \{(y)^1, ((y)^1)^2\}$ joining y to some vertex a of CQ_{n-1}^1 . Then, $P = \langle x, R, (x)^1, ((y)^1)^2, (y)^1, y, S, a \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(2^{n-1} + 3) = y$.

Subcase 3.3. $l = 2^{n-1} + 1$. By Lemma 5, vertices $x, y, (x)^2$, and $(y)^2$ induce a C_4 . By Lemma 4, there exists a Hamiltonian path R of CQ_{n-1}^0 joining x and $(x)^2$, and there exists a Hamiltonian path S of $CQ_{n-1}^1 - \{(y)^2\}$ joining y to some vertex a of CQ_{n-1}^1 . Then, $P = \langle x, R, (x)^2, (y)^2, y, S, a \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(2^{n-1} + 2) = y$.

Subcase 3.4. $5 \leq l \leq 2^{n-1}$. Let $h = l - 2$, and we have $3 \leq h \leq 2^{n-1} - 2$. By Lemma 5, vertices $x, y, (x)^2$, and $(y)^2$ induce a C_4 . By the inductive hypothesis, there exists a Hamiltonian path R of CQ_{n-1}^0 such that $R(1) = x$ and $R(h+1) = (x)^2$. We can write path R as $\langle x, R_1, (x)^2, a, R_2, b \rangle$, where a is a vertex of CQ_{n-1}^0 adjacent to $(x)^2$ and b is some vertex of CQ_{n-1}^0 . Notice that $a = b$ if $h = 2^{n-1} - 2$. By Lemma 4, there exists a Hamiltonian path S of $CQ_{n-1}^1 - \{(y)^2\}$ joining y to $(b)^n$. Then, $P = \langle x, R_1, (x)^2, (y)^2, y, S, (b)^n, b, rev(R_2), a \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(l+1) = y$.

Subcase 3.5. $l = 4$. By Lemma 6, vertices $x, y, (x)^2, ((x)^2)^1$, and $(y)^1$ induce a C_5 . By Corollary 1, there exists a Hamiltonian path R of $CQ_{n-1}^0 - \{x, (x)^2\}$ joining $((x)^2)^1$ to some vertex a of CQ_{n-1}^0 . Path R can be written as $\langle ((x)^2)^1, b, R', a \rangle$, where b is some vertex of CQ_{n-1}^0 adjacent to $((x)^2)^1$. By Lemma 4, there exists a Hamiltonian path S of $CQ_{n-1}^1 - \{(y)^1\}$ joining y to $(a)^n$. Then, $P = \langle x, (x)^2, ((x)^2)^1, (y)^1, y, S, (a)^n, a, rev(R'), b \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(5) = y$.

Subcase 3.6. $l = 3$. By Lemma 5, vertices $x, y, (x)^2$, and $(y)^2$ induce a C_4 . Since CQ_n is $(n-2)$ -fault-tolerant Hamiltonian by Lemma 4, there exists a Hamiltonian cycle C of $CQ_n - \{x, (x)^2, (y)^2\}$. We can write C as $\langle y, R, a, y \rangle$, where a is some vertex adjacent to y . Then, $P = \langle x, (x)^2, (y)^2, y, R, a \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(4) = y$.

Subcase 3.7. $l = 1$. By Lemma 4, there exists a Hamiltonian path R of $CQ_n - \{x\}$ joining y and some vertex a , where $a \neq x$. Then, $P = \langle x, y, R, a \rangle$ is a Hamiltonian path of CQ_n with $P(1) = x$ and $P(2) = y$.

The above completes the proof. \square

4 Concluding remarks

In a hyper-panconnected graph, we can find a path joining any two distinct vertices in a required distance, and this path can be further augmented to form a Hamiltonian path. Since there does not exist any path of length 2 between any two adjacent vertices in the crossed cube, we present a loose version of the hyper-panconnectedness for the crossed cube in this paper. Let x and y be any two distinct vertices of CQ_n . We show that, for any integer l with $d_{CQ_n}(x, y) \leq l \leq 2^n - 1$ and $l \neq d_{CQ_n}(x, y) + 1$, there exists a Hamiltonian path P of CQ_n such that $P(1) = x, P(l+1) = y$, and therefore $d_P(x, y) = l$.

5 References

- [1] S. Abraham, K. Padmanabhan, The Twisted Cube Topology for Multiprocessors: A Study in Network Asymmetry, *J. Parallel Distrib. Comput.* 13 (1991) 104-110.
- [2] Y. Alavi, J. E. Williamson, Panconnected Graphs, *Studia Scientiarum Mathematicarum Hungarica* 10 (1975) 19-22.
- [3] J. A. Bondy, U. S. R. Murty, *Graph Theory*, Springer, London, 2008.
- [4] C. P. Chang, T. Y. Sung, L. H. Hsu, Edge Congestion and Topological Properties of Crossed Cubes, *IEEE Trans. Parallel Distrib. Syst.* 11(1) (2000) 64-80.
- [5] S. A. Choudum, V. Sunitha, Augmented Cubes, *Networks* 40 (2002) 71-84.
- [6] K. Efe, A Variation on the Hypercube with Lower Diameter, *IEEE Trans. Comput.* 40(11) (1991) 1312-1316.
- [7] K. Efe, The Crossed Cube Architecture for Parallel Computing, *IEEE Trans. Parallel Distrib. Syst.* 3(5) (1992) 513-524.
- [8] J. Fan, X. Lin, X. Jia, Optimal Path Embeddings in Crossed Cubes, *IEEE Trans. Parallel Distrib. Syst.* 16(2) (2004) 1190-1200.
- [9] J. Fan, X. Lin, X. Jia, Node-pancyclicity and Edge-pancyclicity of Crossed Cubes, *Inf. Process. Lett.* 93 (2005) 133-138.
- [10] J. Fan, X. Jia, X. Lin, Complete Path Embeddings in Crossed Cubes, *Inf. Sci.* 176 (2006) 3332-3346.
- [11] W. T. Huang, Y. C. Chuang, J. J. M. Tan, L. H. Hsu, On the Fault-tolerant Hamiltonicity of Faulty Crossed Cubes, *IEICE Trans. Fundamentals* E85-A(6) (2002) 1359-1370.
- [12] P. Kulasinghe, S. Bettayeb, Embedding Binary Tree into Crossed Cubes, *IEEE Trans. Comput.* 44(7) (1995) 923-929.
- [13] J. H. Park, H. C. Kim, H. S. Lim, Many-to-many Disjoint Path Covers in Hypercube-like Interconnection Networks with Faulty Elements, *IEEE Trans. Parallel Distrib. Syst.* 17(3) (2006) 227-240.
- [14] Y. Saad, M. H. Shultz, Topological Properties of Hypercubes, *IEEE Trans. Comput.* 37 (1988) 867-872.
- [15] X. Yang, D. J. Evans, G. M. Megson, The Locally Twisted Cubes, *Int. J. Comput. Math.* 82(4) (2005) 401-413.

Modification and Evaluation of Software-Based Communications Unit of a LSC on Chip

Akiko Narita, Naoya Kato, Kenji Ichijo, and Yoshio Yoshioka

Department of Electronics and Information Technology, Hirosaki University, Hirosaki, Aomori, Japan

Abstract - We have evolved several types of Loop Structured Computers (LSCs) to build a system with small hardware overhead to apply it system-on-chip (SoC). In recent studies, we investigated control flow methods with software-based approach with a general-purpose instruction set. In this paper, we proposed two specialized instruction sets for communications in order to reduce transmission latency. One used similar CPU design to the previous study. The other required support of hardware pointers and counters. We evaluated performance of the new systems with simulations with clock cycle level of the CPU. The former instruction set shortened minimum packet transmission latency only a few percent under low traffic, and improved throughput about 10 %. The latter declined minimum packet transmission latency about 20% in number of clock cycles, and increased throughput 30 %. Furthermore, size of communications program was shrieked about 30 %.

Keywords: Multiprocessor system; parallel processing; system-on-chip; uni-directional loop

1 Introduction

Processor interconnection methods for parallel computers such as topology, routing, flow control, etc. are important issues. They have effect on performance and cost of them. Huge and expensive hardware resource is allowed to spend to achieve high performance for a high-end system. On the other hand, in system-on-chip (SoC) of an embedded system, each module must be constructed under restrictions. They are size, power saving, cost, and so on.

Small size of communications units (CUs) for connecting processing units (PEs) are required to resolve hardware restrictions of SoC's design problem. In [1], lightweight routers models for a bi-directional ring network were proposed. We have evolved various types of Loop Structured Computers (LSCs) [2]-[6]. The LSC is a multi-processor system designed for data flow processing. All processing elements in it are interconnected one another uni-directionally with point-to-point connection. Indeed a loop of PEs is constructed. Plane topology of the LSC releases the CU from complicated routing. It decides only whether if it takes or hops a packet. This simplest topology enabled to implement dynamic load balancing algorithm with a simple

method and meet high scalability requirements [7]. This topology also has flexibility for extending or reducing system size. Moreover, this feature presented redundancy for reliability [8]. Thus, LSC has preferable characteristics to construct a multi-processor system and configure network-on-chip [9] with limitation of hardware resources.

At the same time, one of defects of the LSC is transmission latency for large system. It increases proportionally to system size for topological characteristic. In spite of the defect, the ring topology exhibited good transmission performance for specific traffic [10] [11]. If execution time of a task assigned to the PE is large, it cancels overhead of the packet transmission. The ring topology is not popular in current SoC [12], [13] and not sufficient research has been done about possibility of this topology.

In early works of the LSCs, the PE emulated the CU. Then store-and-forward method was adopted for packet flow control in spite of large transmission latency. Advantage of the method was small overhead of communications procedures executed by the PE. However, it is appreciable to reduce packet transmission latency. The shorter overhead extends applicable service of the parallel processing system. Therefore, we separated the CU from the PE and investigated appropriate packet flow control methods [14] [15]. Software-based approach was applied. Simulations with clock cycle level of the CPU of the CU were performed for evaluation. The simulations could provide more accurate results than ordinary researches evaluating network performance in which flit level simulations were carried out. We obtained results that virtual cut-through method [16] was superior to the other methods for packet transmission latency and throughput. While the wormhole routing method [17] is a dominant one in current SoCs [12] [13], its complicated control procedure has hindered performance.

On the other hand, the CPU utilized in [14] [15] was not optimized for the purpose of communications. Then we tried to propose new instruction sets, which were suitable for a communications program in order to reduce transmission latency. Two instruction sets were applied in this study. Both of them included specialized instructions for flit receiving and sending procedures. One could be implemented without drastic modification of the previous CPU. The other required support of hardware pointers and counters. We evaluated the communications systems with proposed instruction sets with

clock cycle level simulations. This study exposed probability and limitation in software-based approach that consumes small hardware in constructing the CU of the LSC.

2 Proposed system

2.1 Overview of the LSC

In this section, we illustrate an outline of the LSC. Fig. 1 shows basic structure of the LSC. The PEs are interconnected with the CUs, which send a packet unidirectionally with buffered flow control. PE₀ communicates with a host computer or other modules. On the other hand, PEs execute application programs exchanging the packets one another. Packet format is given by Fig. 2. Suitable data length *m* of the packet was chosen for applications. Fig. 3 presents basic structure of the CU. Its functions are implemented with a CPU for saving hardware consumption and flexibility of procedures. A communications program including instructions, communications buffers, and control variables are stored in a memory unit. A registers for transmitting data and status in a communications interface are accessible from the communications program with memory mapped I/O. The communications interface transmits asynchronously with an 8-bit data lines and handshake lines of strobe (STRB) and acknowledgement (ACK).

2.2 Modified instruction set and the CPU

The instruction set in the previous studies [13] [14] was designed for general purpose in order to investigate control flow method flexibly. Number of clock cycles for transmission was expected to decline with an optimized instruction set for role of the CU. We proposed new instruction sets suitable for the communications program in order to explore probability and limitation of our approach.

We applied two instruction sets. One was set A that could be implemented without drastic modification of the previous CPU. Several paths were added and control unit was modified for implementation. The other was set B that needed hardware pointers and counters. Table I shows the instruction sets. The first 16 instructions are original ones included also for the previous system. The others are new instructions. They were prepared for each counter or pointer used in the flit transmission procedures shown in Fig. 4. Only extended load and store instructions were added to the set A.

We focused on flit transmission procedures to customize the new instruction sets. There were two receiving channels and sending channels for the CU as shown in Fig. 3. Two flit buffers were prepared for receiving. A receiving channel was linked a target channel of sending before packet sending started. There were several attentive points. Firstly, the procedures in Fig. 4 were executed repeatedly. Secondly, access to the flit buffer was accompanied with update of counters as shown in R5 and S6 in Fig. 4. Furthermore, counter values were changed only in these steps. Thirdly,

pointers in the step R3 and S3 in Fig. 4 were memory data in the previous study. Namely, addressing mode of the load and store instructions used in the steps was memory indirect. These instructions spent about twice clock cycles of the other instructions as shown in Table I. Lastly, the counters were variables in the memory.

To reduce clock cycles spent in access to the flit buffer, we merged instructions used for steps of R3, R4 and R5 in Fig. 4. Thereby the extended store instructions were built. In other words, the extended store instructions were compound of store and three times of counter increment. Likely, instructions used for S3, S4, and S6 were combined as the extended load instructions. They were composed of load, two times of counter increment and decrement. The instruction set

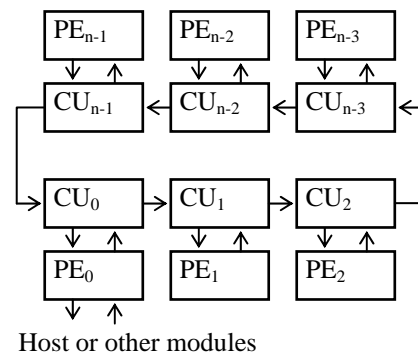


Figure 1. Basic structure of the LSC.

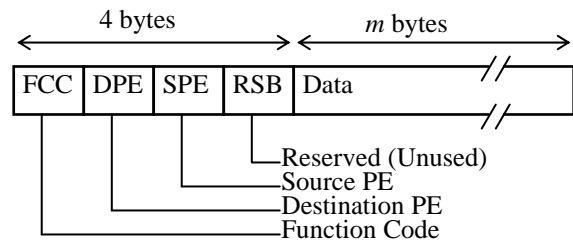


Figure 2. Packet format.

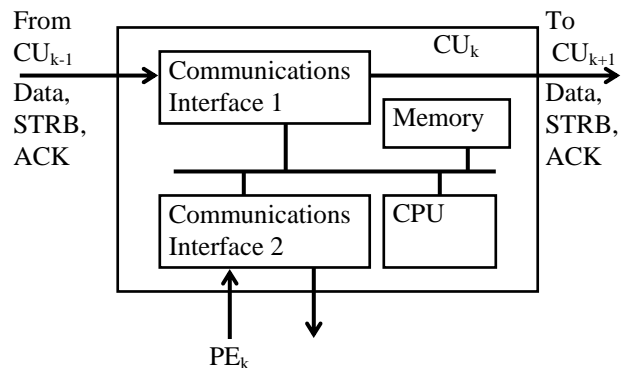


Figure 3. Structure of the CU.

TABLE I INSTRUCTION SET OF THE COMMUNICATIONS CPU.

Instructions	Type *)	Execution time [clocks]
No operateion	I	1
Load(immidiate)	II	2
Load(direct)	III	3
Load(memory indirect)	IV	7
Store(direct)	III	3
Store(memory indirect)	IV	8
Bit Test and jump	IV'	4
Compare and jump	IV'	4
Jump	III	3
Increment memory data	II	4
And(immidiate)	II	3
Or(immidate)	II	3
Add(immidate)	II	3
Add(dorect)	III	4
Exclusive or(direct)	II	4
Rotate right	I	3
Load(immidate 11bits)	B III'	2
Load(extended)	A I	3
	B V'	15
Store(extended)	A I	3
	B V	13
Register data transfer	B I	2
Increment register data	B I	2

*) Type is shown in Fig. 6.

A consisted of original instructions and only these extended instructions. These new instructions restricted location of variables. Most significant 3 bits of addresses of reading/writing pointers and counters for the same flit buffer must have been the same value as of the flit buffer.

The instruction set B was designed to reduce clock cycles more aggressively than A decreasing frequency of memory access. We put the pointers of the flit buffers on registers in the CPU. Furthermore, counter variables were taken in hardware counters in the CPU. Accordingly, steps for counter check of R1, R6, S1, and S2 referred register data but not memory data. The register transfer instructions for counters in Table I were arranged for these steps of checking counters. Instructions of loading 11-bit data and incrementing register data enabled to describe communications program effectively using the instruction set B.

Format of the instructions are shown in Fig. 5. Increase of number of instructions imposed of the instruction formats partially. Length of operation codes (OPs) was 5 bits in the proposed instructions while 4 bits in the previous ones. This extension of the length of the operation code took one bit away from the address field. Therefore, effective size of memory shrieked from 4096 bytes to 2048 bytes, which was sufficient for the present CU. 8-bit address fields of format

Receiving:

- R1. Check a flit buffer if it is not full referring a counter of waiting flit,
- R2. Receive a flit,
- R3. Store the flit in the flit buffer referring a pointer for writing,
- R4. Increment the pointer,
- R5. Increment counters of received flits and waiting flits,
- R6. Check the flit if it is DPE referring the counter of received flits and call channel link procedure if required.

Sending:

- S1. Check a flit buffer if it is not empty referring a counter of waiting flit,
- S2. Chick a counter of sent bytes and call initializing procedure to start sending a packet if required,
- S3. Load a flit from the flit buffer referring a pointer for reading,
- S4. Increment the pointer,
- S5. send the flit
- S6. Increment the counter of sent flits and decrement that of waiting flits.

Figure 4. Common procedure for receiving/sending a flit.

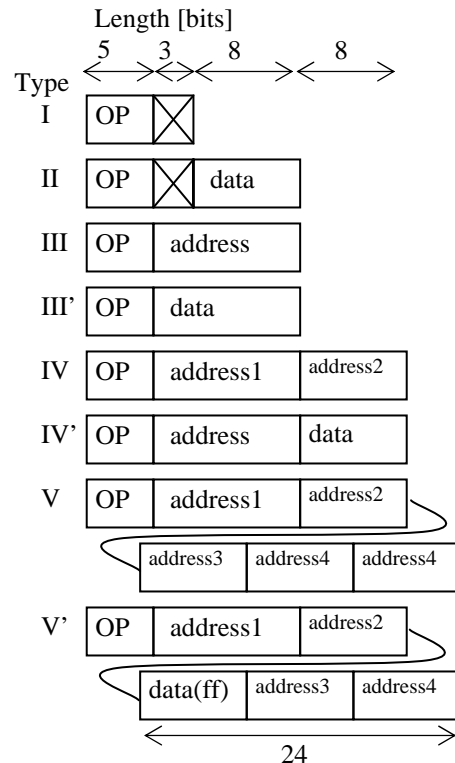


Figure 5. Format of instructions.

type IV, V, and V' mean least significant 8 bits of 11-bit addresses. Most significant 3 bits of addresses are common with the former 11-bit address fields. Address operands of the format type V are addresses of a received counter, a waiting flit counter, a write pointer for a flit buffers, respectively. The last two fields must have the same value. Since there is insufficient space in the CPU to hold "address4" during processing of the instruction, it must be given twice an instruction. Address operands of the format type V' are addresses of a sent flit counter, a waiting flit counter, a read pointer for flit buffers, respectively. Data filed is for counter decrement. Since an ALU in the CPU did not have function of subtraction, down count was realized by addition of value ff.

Fig. 6 presents architecture of the CPU, which realize the instruction set B. Gray colored counters and registers are added devices to the previous CPU. We described the CPU

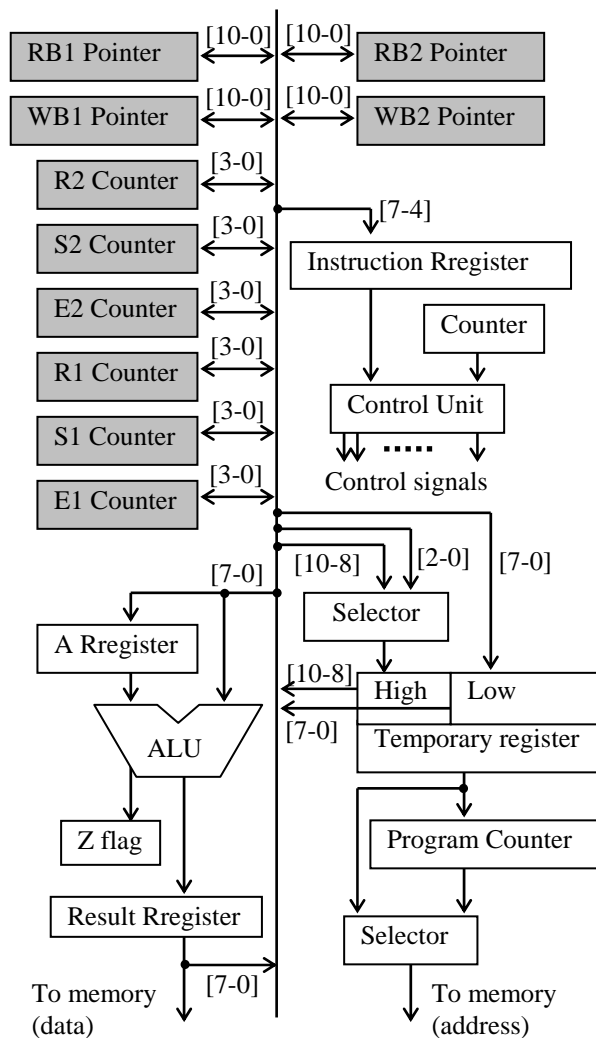


Figure 6. Architecture of the proposed CPU.

with very high-speed integrated circuit hardware description language (VHDL). Codes of the CPU were compiled by Quartus II 10.0sp1 Web Edition for the target device of a field programmable gate array (FPGA) EPM3512AQC208-7 of MAX3000A series of Altera. Then simulation was done and confirmed to be correct design.

Fig. 7 and 8 shows local effect of the proposed instruction set for execution time and program size. The instruction set B could drastically reduce both number of clock cycles and size of routine in the flit transmission procedures in Fig. 4. Effect of the instruction set A was less than that of the set B.

3 Evaluation

3.1 Latency and throughput

Improvement of the latency and throughput of the system using proposed instruction sets were evaluated with simulations. Target method of flow control applied in this paper was virtual cut-through method since it had given the best performance in the previous study [13] [14]. Note that the new instruction sets did not prevent executing the other flow control method. To evaluate proposed system, we rewrote communications programs with the new instruction sets. The simulator was described with C language. It traced status of the CU by the clock cycle of the CPU. The status

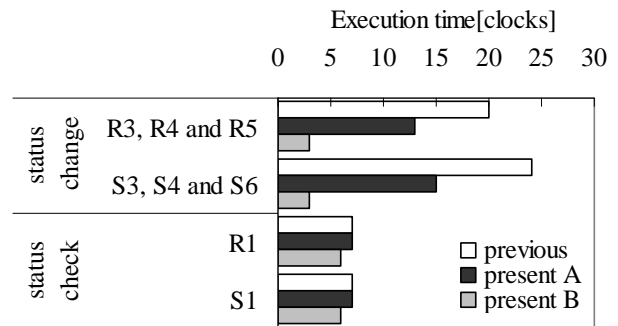


Figure 7. Local decrease of number of clock cycles.

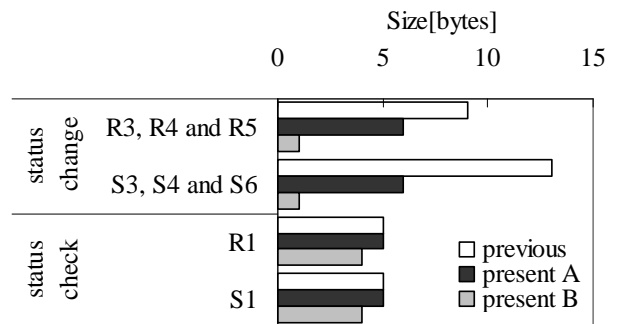


Figure 8. Local decrease of program size.

consisted of data in packet or flit buffers, an instruction executing, variables in the communication program, data and validity of them in a communications interface, and a signal on a handshake line. To deduce performance of the CU independently of the performance of the PE, very short intervals of flit transmission by the PE were assumed. In the simulations here, data size of packet m was 8.

Fig. 9 shows the minimum latency. If traffic is so low that all CUs service at most one packet simultaneously, CUs devote their capability to transmission of the packet and minimum latency was obtained. A packet delivered to PE_k was generated in PE_0 in each simulation in which PE-PE distance equaled to k . The latency consisted of three elements. They are latency of flit transmissions between a source PE and a CU attached to it, among CUs, and between a destination PE and its entrance CU, respectively. The delay of CU-CU transmission increased by hop proportionally to k as expectedly. This element became dominant for large k . The CU-CU transmission was accomplished in 20% smaller number of clock cycles with the instruction set B than with the previous system, while only a few percent of improvement was obtained with the set A.

Under frequent request of packet transmission, the CU must manage both communications with the PE and the other CUs. Accordingly, the CU spends less time for a certain packet in ratio of execution time of the CPU. Then packet transfer latency for a particular packet increases with traffic while throughput improved. To compare performance under traffic, we simulated transmission with Poisson arrival process of packet generation of the PE for a 64-PE system. Distances between source and destination PEs distributed from 1 to 63 uniformly. 32 was an average distance of them. Simulation duration was 100 times longer than average packet generation intervals so that each PE generated, sent, and received about 100 packets if the traffic was enough low to accept for the system. Packet transmission latency under uniform traffic and throughput are shown in Fig. 10. The throughput is defined as number of packets that arrive at

destination per clock cycle.

Packet transmission latency per packet measured with number of clock cycles was always the smallest with the proposed instruction set B. The set A gave less latency than that of the previous study. Difference between them became larger with traffic. That is to say, the communications program with the present study is more advantageous under high traffic. If the traffic was low, most of execution time of the CPU was spent for testing whether if a flit had arrived. The proposed instruction set had no profit for this procedure. If the traffic was high, the CU carried out flit transmission shown in Fig. 4 frequently and contribution of reduction of clock cycles explained in the section 2.2 appeared apparently. In spite of little effect under the low traffic, the set A declined packet transmission latency about 10% compared to the previous system under the high traffic.

When the traffic was low, both the previous and present systems could accept offered traffic completely so that the throughput was the same for them. However, the traffic increased, the throughput of the previous system saturated first. If the system cannot deliver packets as fast as rate of packet generation continuously, buffers in CUs and PEs are filled with the packets and the system cannot improve the programs. Therefore, effective range of the packet generation with the instruction set A and B were about 10 % and 30 % wider than that of the previous system.

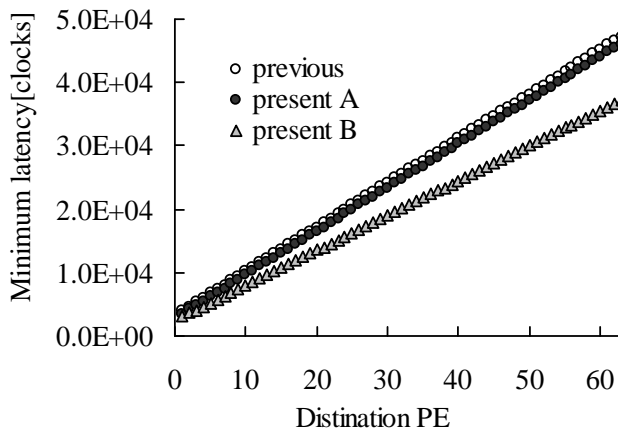


Figure 9. Minimum latency.

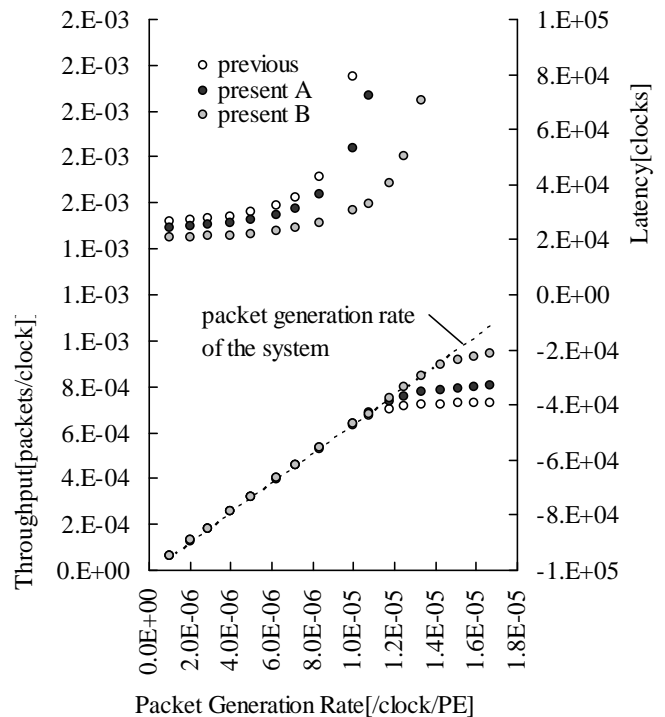


Figure 10. Latency and throughput under uniform traffic.

3.2 Program Size

Some of the proposed instructions were designed by merging the previous instructions. This composition compressed size of the transmission procedure smaller as shown in Fig. 8. As a result, size of communications programs with the new instruction sets were decreased although a few temporary variables were added to rewrite the communications programs effectively. Sizes of the programs are shown in Fig. 11. Using the instruction sets A and B, amount of the communications program sizes were decreased about 10 % and 30 % of that of previous program, respectively. The instruction set B lessened not only instruction area but also variable area on the memory since the pointers and counters were moved to the CPU.

4 Conclusions

Developing the LSC, we modified the instruction set of the CPU in the CU in order to optimize software-based communications. Two instruction sets of A and B were proposed to decline number of clock cycles in execution of flit transmission procedures. The instruction set A was built only merging some of original instructions that were used consecutively. The instruction set B was constructed to reduce frequency of memory accesses with support of the hardware pointers and counters in the CPU. We also designed the new CPU to implement the instruction set B. Then we rewrote communications program with virtual cut-through method for flow control and compared the new systems to the previous one with simulations. The simulations were performed with clock cycle level of the CPU. The proposed instruction set B decreased packet transmission latency under low traffic about 20% in clock cycles, while the set A gave little improvement. The proposed instruction sets were more advantageous under higher traffic. Using the instruction set A and B, maximum throughputs were about 10 % and 30 % higher than the previous one, respectively. Furthermore, size of communications program shrieked about 10 % and 30 % with the set A and B, respectively.

Consequently, it is significant to reduce memory access with support of hardware to decline execution clock cycles of packet transmission. On the other hand, if the LSC is used

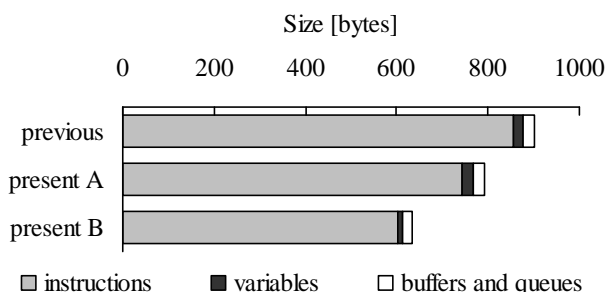


Figure 11. Program size.

under high traffic, it is considerable to implement merged instructions optimized for transmission purpose even if with simple hardware.

In future work, advanced optimization of instruction set must be carried out observing trade-off of hardware and software in detail. When we compiled VHDL code of the CPU for the instruction set B for the target device EPM3512AQC208-7, number of required macro cells and clock set up were about twice of the previous system. These results were no more than an example under restriction of a particular FPGA chip. Moreover, only the CPU was complied so that effect of decrease of size of memory was not included. If our system including memory and the other devices are implemented as custom ICs, different results should be obtained. However, these preliminary results suggest importance of consideration for effect of hardware complexity applying sophisticated method.

5 References

- [1] John Kim and Hanjoon Kim. "Router microarchitecture and scalability of ring topology in on-chip networks". Proc. 2nd International Workshop on Network on Chip Architectures, Dec. 2009, pp. 5-10.
- [2] YOSHIOKA, Y. "Constructions of the Loop Structured Computer"; SCi. Rep. Hirosaki Univ. Vol. 41, no. 1, 157-172, Aug 1994.
- [3] ISHIDA, T., NARITA, A., YOSHIOKA, Y. "Construction of a Donut Type of Loop Structured Computer and its Characteristics". SCi. Rep. Hirosaki Univ., Vol. 42, no. 1, 147-156, Aug 1995.
- [4] Y. Yoshioka and A. Narita. "The data flow processing by the LSC with broadcast mode"; The Science Reports of the Hirosaki University, 43-2, 289-299, Dec 1996.
- [5] Y. Yoshioka and D. Zhao. "The Functional Distributed Control System Using a Dataflow Processing Scheme and its Traffic Characteristics"; TECHNICAL REPORT OF IEICE, NS2002-9, 33-36, Apr 2002.
- [6] A. Narita, K. Ichijo and Y. Yoshioka. "The Parallel Computer LSC for Simulations of the Self-Reproduction Type Model"; INFORMATION, Vol. 12, No. 3, pp.663-672, May 2009.
- [7] A. Narita, X. Zhao, S. Mizuta, and Y. Yoshioka. "Scalability of the LSC for Simulation of Self-Reproduction Model"; Proc. the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), 514-520, Jun 2001.

- [8] M. Fukushi, A. Narita, S. Mizuta, Y. Yoshioka, S. Horiguti. "A fault tolerant method for the Loop Structured Computer". TECHNICAL REPORT OF IEICE, FTS99-32, Aug 1999, pp. 7-14.
- [9] L. Benini and G. De Micheli. "Networks on Chips: A New SoC Paradigm"; Computer, vol. 35, no. 1, pp.70-78, Jan 2002.
- [10] Michael Kistler, Michael Perrone, Fabrizio Petrini. "Cell multiprocessor communication network: built for speed". IEEE Micro, v.26 n.3, p.10-23, May 2006
- [11] K. Lahiri, S. Dey, and A. Raghunathan. "Evaluation of the traffic-performance characteristics of system-on-chip communication architectures". Proc. 14th International Conference on VLSI Design, Jan. 2001, pp. 29-35.
- [12] T. Bjerregaard and S. Mahadevan. "A survey of research and practices of network-on-chip". ACM Computing Surveys, vol. 38, 2006, pp. 1.
- [13] E. Salminen, A. Kulmala, and Timo D. Hamalainen. "Survey of network-on-chip proposals". White Paper, OCP-IP, March 2008.
- [14] Akiko Narita, Kenji Ichijo and Yoshio Yoshioka. "Evaluation of Packet Flow Control Methods for a LSC on Chip with Hardware Requirements and Performance". Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2010), pp.390-396
- [15] Akiko Narita, Kenji Ichijo and Yoshio Yoshioka. "Comprehensive Evaluation of Packet Flow Control Methods for a Ring Network of Processors on Chip". Proc. of the 9th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2010), pp. 75-8
- [16] O. Kermani and L. Kleinrock. "Virtual cut-through: a new computer communication switching techniques". Computer Networks, vol. 3, No. 4, 1979, pp. 267-286.
- [17] L. M. Ni and P. K. McKinley. "A survey of wormhole routing technique in direct networks". Computer, vol. 81, No.2, Feb. 1993 pp. 62-76.

A Protocol for Realtime Switched Communication in FPGA Clusters

Richard D. Anderson

Computer Science and Engineering, Box 9637
Mississippi State University
Mississippi State, MS 39762
rda62@msstate.edu

Yoginder S. Dandass*

Computer Science and Engineering, Box 9637
Mississippi State University
Mississippi State, MS 39762
yogi@cse.msstate.edu

Abstract - *Field programmable gate array (FPGA) devices typically have limited resources. This means that networks of FPGAs are required for implementing large-scale applications. Use of realtime communication channels can be used for reducing handshaking overhead in order to create high-performance networks. This paper describes a switched, real-time, link-level protocol and its implementation using Virtex-4 multigigabit transceivers. A prototype 4Gbps implementation of the protocol shows a per-hop latency and jitter of 310ns and 12.5ns, respectively, when endpoints run at a clock frequency of 100MHz. The prototype is also used to demonstrate the implementation of a jitter-free distributed global realtime clock that can be used for scheduling communication and computation of applications implemented using this cluster of FPGAs.*

Keywords: Cluster, FPGA, Real-time, Multi-gigabit transceivers, communication protocol.

1 Introduction

Reconfigurable computing, using field programmable gate arrays (FPGAs), is becoming increasingly popular in high-performance computing applications such as in digital signal processing (DSP) and in bioinformatics. FPGA-based applications exploit the massive parallelism that can be implemented in logic. However, the level of parallelism that can be implemented is often restricted by the area (i.e., resources) available on the chip. Furthermore, large FPGAs are significantly more expensive as compared to smaller chips. Therefore, in many applications it is more cost effective to utilize a number of smaller interconnected FPGAs than using one (or more) large FPGAs.

We had previously introduced the design of a real-time link-level communication protocol and a prototype point-to-point implementation using multi-gigabit transceivers (MGTs) found on Virtex-4 FPGAs [2]. Although this previous

implementation had good real-time characteristics, our prototype was limited to point-to-point communication between exactly two FPGAs. This paper extends the ideas introduced previously by developing and characterizing a switch that enables routing of frames in real-time between several FPGAs. Furthermore, our enhanced design improves the real-time performance of the protocol by reducing the jitter of the real-time distributed clock. Reducing this jitter is important for reducing the overheads in a real-time network. The remainder of the paper is organized as follows: Section 2 discusses the motivation and background for this work. Section 3 provides a summary introduction to the protocol. Section 4 describes the experimental setup and results, and Section 5 concludes with a discussion the results and future work.

2 Background and Motivation

Many FPGA-based applications exhibit real-time characteristics because their applications' logic are controlled by finite state machines (FSMs) having well-defined timing properties. When designing a communication protocol for such applications distributed over a cluster of FPGAs, the handshaking required for flow control between devices can be eliminated. This results in a high-performance network in which communication is scheduled according to a globally distributed real-time clock.

Using the protocol presented here, applications typically utilize zero-sided communication as opposed to two-sided communication found in most software applications. In two sided communication, processes at both ends of the communication channel execute communication operations in order to transfer data. In zero-sided communication, neither endpoint process issue explicit data transfer operations. Instead, any data that is available in a buffer at the transmit endpoint is delivered to a buffer at the receiving endpoint according to a predefined schedule.

*Corresponding Author

Modern FPGAs such as Xilinx's Virtex [3], [4] and Altera's Stratix [5] families of FPGAs support several multigigabit transceivers that can be used for chip-to-chip or board-to-board serial communication. Xilinx has designed a general-purpose high-speed serial communication protocol, Aurora, for enabling simplex and full-duplex link-level communication in applications using on-chip MGTs [6]. Aurora supports both frame-oriented and streaming communication with native and application controlled flow control. Xilinx's Aurora implementations are feature-rich and robust. However, we chose to design our own protocol in order to reduce latencies and overheads to the largest extent possible.

A few research projects have also developed and implemented customized point-to-point high-speed communication channels for interconnecting FPGAs using MGTs [1], [7], [8]. The Aurora protocol is also relatively efficient but is a point-to-point protocol. Because of the limited number of MGTs available at each FPGA, creating large clusters requires the use of interconnect switches. Aurora does not have switching capability, and therefore, any frame routing capability needs to be developed as part of the application.

3 Protocol Description

We have adapted and enhanced the implementation of the protocol described in detail in [1], and therefore, do not repeat all the protocol details here in order to conserve space. However, important information on the specification of routing and its implementation in our new version is described in detail. Our protocol is a link-level serial protocol that uses 8b/10b encoding [9] to exchange control and data messages between application endpoint and switch FPGAs. Only frame-oriented communication is supported and endpoint FPGAs are connected to each other using switches over point-to-point physical links. In its present form, the protocol allows a maximum of eight intermediate switches between any two nodes and each switch can have at most eight connections to other endpoints and switches.

Just as in [1], our protocol uses source routing (i.e., the application endpoint determines the routing of each frame and includes the routing information at the beginning of each frame). The switches use the routing information in order to perform cut-through (or worm-hole) routing. In cut-through routing, the switch forwards data words as soon as they are received and the outbound channel is available. This significantly decreases latencies as compared with traditional store-and-forward routing.

3.1 Frame Layout and Routing

Data and control bytes are transmitted in 4-byte words. Frames are delimited by 4-byte *start-of-frame* (SoF) and *end-of-frame* (EoF) control words. When no data is available for transmission, the MGT sends the *idle* control word. Idle control words are continuously transmitted between frames in order to maintain the synchronization between a pair of communicating serial transceivers (i.e., the receiver extracts the clock embedded in the transitions between 0 and 1 signals in the physical data stream being sent by the transmitter). Therefore, the receiver logic on one endpoint FPGA is operating at the same frequency as the transmitter logic on the switch FPGA.

It is also possible for the transmitter to run out of data to send before the frame is completely transmitted. In this case, the transmitter also sends idle control words within the frame; these idle control words are discarded by the receiver. Note: if the real-time schedule is constructed carefully, there should be no need to transmit idle control words within a frame. However, this feature was designed into our protocol to provide some safety against data buffer underflows in case there are small differences in frequencies between the transmitter's, receiver's, and application's clock domains.

The SoF word is immediately followed by a single source routing word. Source routing information is organized into 8 nybbles (i.e., 4-bit sequences) such that each nybble represents a single hop (i.e., an outgoing transmission from a switch). The most significant bit of a hop specification indicates whether the remaining three bits are valid. The remaining three bits specify the outgoing port on the switch that is to be used to forward the frame. Essentially, the value in the three address bits is added to the incoming port number to derive the outgoing port number. The complete route for a frame is specified by the sequence of nybbles starting with the least significant nybble in the word. When the source routing word is received by a switch, it scans the word starting with the least significant nybble looking for a valid hop specification (invalid specifications are skipped). When a switch forwards the frame, it consumes the corresponding nybble by clearing the valid bit in the nybble (see [1] for more details).

For data integrity checks, an optional 4-byte error-checking word (e.g. checksum or CRC) can be inserted into the frame immediately before the EoF. The switch can be configured to detect this word, verify the integrity of the frame payload, and invalidate an erroneous frame.

3.2 Implementation Issues

We have developed a prototype implementation of the link layer logic on the Virtex-4 FX family of FPGAs. In our implementation, each Virtex-4 MGT has different clocks for the transmitter and the receiver. An on-board reference oscillator is used for clocking the transmitter whereas the receiver's clock is derived from the received signal. Furthermore, the application on the endpoints has its own independent clock. The MGT can simplify implementation clocking by performing synchronization internally such that the link layer needs to only be aware of one clock domain. However, internal clock synchronization requires the data and control words to traverse through a fairly lengthy internal path [1]. A significant portion of this path can be bypassed by operating the MGT in a "low latency" mode with the tradeoff that the link layer must perform synchronization by itself.

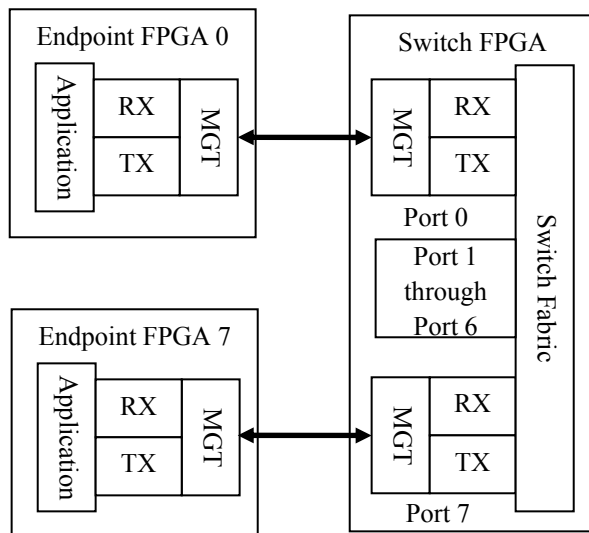


Figure 1: Implementation Architecture [1]

In our prototype implementation, the switch uses eight receivers and eight transmitters. All of the eight transmitters derive their clocks from a common 100MHz reference clock. However, because each of the eight receivers derives its frequency from the transmitter it is connected to, each of the eight receivers in the switch operates in an independent 100MHz clock domain. Similarly, on the endpoints, the application, transmitter, and receiver each operate in independent 100MHz clock domains.

We use FIFOs with independent read and write clocks whenever data needs to cross clock domains. Xilinx supports efficient construction of such FIFOs using block RAMs (BRAMs) in Virtex 4 FPGAs [3]. These FIFOs, however,

each add five to six clock cycles of latency in the data path (the variance is caused by the phase relationship of the clocks at FIFO's read and write endpoints). Therefore, significant latency can result when data traverses several clock domains. For example, when data is transmitted from the application block in *Endpoint FPGA 0* to the application block in *Endpoint FPGA 7*, it must traverse three FIFOs (Application-to-TX at Endpoint FPGA 0, RX_{port 0}-to-TX_{port 7} at the switch, and RX-to-Application in FPGA 7).

3.3 Implementation Architecture

Figure 1 depicts the architecture of the link layer. It shows one endpoint FPGA connected to the switch FPGA. The host FPGA uses one MGT to connect to the switch. The switch FPGA uses eight MGTs to connect to other switches or endpoint FPGAs. The internal signal routing resources of the switch FPGA are used to create a fully connected point-to-point topology between the switch's MGTs.

The switch does not provide buffering services greater than the depth of the FIFOs between the ports; it is the application's responsibility to create frames and transmission schedules such that it does not overflow the FIFO. We chose a fully connected topology to simplify switching and to avoid loss of data due to frame collisions. A collision occurs at a switch when two (or more) overlapping frames are received that need to be sent out on the same port simultaneously. In the event of a collision, our switch receives frames destined for the same port and buffers them in the separate FIFOs connecting the receiver ports with the transmitter port. It then uses a fixed priority scheme to select the order in which the frames will be forwarded. Although the switch is capable of handling a few arbitrary frame collisions, it is not designed to handle constant simultaneous frame transmissions that are destined for the same port. Therefore, it is also the responsibility of the application to create schedules that avoid collisions.

3.4 Switching and Selection Unit Architecture

Frames received by the switch are handled by two hardware modules, the switching unit and the selection unit. There are eight switching units and eight selection units (i.e., one pair per MGT in the switch). Each switching unit is connected to all eight selection units. This results in a total of 64 connections (including loopback connections). Buffering FIFOs are placed between the switching and selection units. All switching units and selection units are independent of each other and can operate in parallel, therefore, it is possible

for all switching units to route frames and for all selection units to forward data concurrently.

The switching unit is responsible for decoding a received frame's source-routing word and then forwarding the frame to the corresponding destination. The switching unit functions as a one-to-many de-multiplexer, where the decoded routing nybble acts as the selector that connects a single input line from the receiver to one of many output lines. A secondary function of the switching unit is to verify the integrity of the frame data using a verification method such as the cyclic redundancy check (CRC). After decoding the route, the switching unit begins computing the CRC for the incoming frame payload. The frame CRC is stored in the 4-byte word at the end of the payload and before the EoF. Once the switching unit sees the EoF, it will verify its computed CRC with the frame CRC and mark the frame as invalid if there is not a match. To ensure that the switching unit properly detects the source-routing word and the frame CRC, the endpoint applications must follow that rule that no idle control words should be placed between the SoF and the source-routing word, and between the CRC and EoF. These word pairs must not be separated.

The switching unit supports zero-sided communication and does not throttle incoming frames; it consumes and forwards each 4-byte data word received by the MGT receiver every clock cycle. However, since the routing information follows SoF in our frame layout, the SoF is buffered in a two stage pipeline to give the switching unit time to decode, update, and register the routing nybble before forwarding the frame; the pipeline is also used when verifying the CRC.

Figure 2 shows a frame being routed by the switching unit. The data flow across the switching unit begins with the receipt of the SoF from the receiver at clock cycle 0. At clock cycle 1, the SoF has been registered in pipeline stage one and the 4-byte routing information is being presented by the receiver. At this point, SoF is assigned to be registered at pipeline stage two and the least-significant valid routing nybble is decoded and assigned to be registered on the next clock cycle. The updated 4-byte routing information, where the decoded nybble's valid bit is set to zero, is assigned to be registered in pipeline stage one. At clock cycle 2, the registered SoF in pipeline stage two is written to the FIFO input port pointed to by the registered nybble, the updated routing information is assigned to pipeline stage two, and the first 4-byte payload is assigned to pipeline stage one.

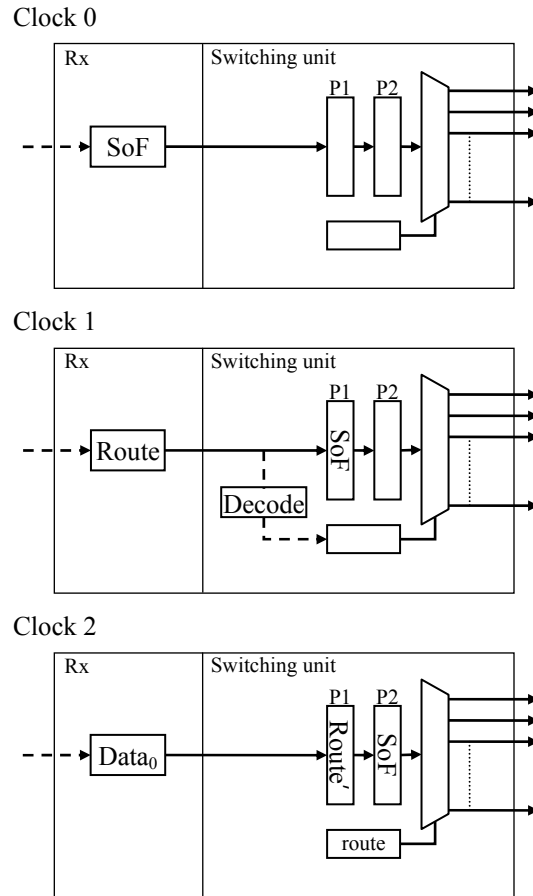


Figure 2: Switching Unit Data Flow

From this point on, each incoming 4-byte payload traverses the pipeline and is inserted into the FIFO until the EoF is presented by the receiver at clock cycle n , where n is the (*size-of-the-frame* - 1), in words. At this point, if the switch is configured to do error checking, the frame's 4-byte CRC has already been registered in pipeline stage one. Upon detecting the EoF, the switching unit compares its computed CRC with the frame's CRC. If the two do not match, the switching unit marks the frame CRC as a bad CRC and sets its value to be registered in pipeline stage two at clock cycle $n+1$; if the CRCs match, the frame CRC is just shifted to pipeline stage two. At clock cycle $n+1$, EoF has been registered in pipeline stage one and the frame CRC is written to the FIFO from pipeline stage two. At clock cycle $n+2$, the EoF has been registered in pipeline stage two and written to the FIFO. The registered routing nybble is flagged to be reset on the next clock cycle.

The selection unit (or selector) implements a many-to-one connection. It receives data from eight FIFOs and forwards their data to the transmitter by selecting a non-empty FIFO

and linking its data-out port and read enable signal to the transmitter. The transmitter can then assert the read enable of the FIFO when it is ready to send the contained frame.

A typical selection process begins when at least one *FIFO-is-empty* indicator is deasserted (e.g. transitions from high to low); all eight *FIFO-is-empty* indicators are observed in parallel. For this implementation, we have used a simple round-robin priority scheme to select between competing non-empty FIFOs. The FIFO with priority is checked for data (i.e. not empty) at this time. If it is not empty, the FIFO is selected over all other non-empty FIFOs and will be linked to the transmitter. The selection unit will also link the *FIFO-is-empty* indicator of the selected FIFO to the transmitter to signal that there is data available to be transmitted. The selector then sets the next sequential FIFO as the new FIFO with priority. For example, if FIFO₃ has priority and is not empty, the selection unit will link it to the transmitter and then set FIFO₄ as the next FIFO with priority. In the event that the FIFO with priority is empty, the next sequential non-empty FIFO is selected. For example, if FIFO₃ is empty and the next non-empty FIFOs are FIFO₆ and FIFO₇, FIFO₆ will be linked to the transmitter and FIFO₇ is set as the new priority. When the round robin reaches FIFO₇, it will wrap around to FIFO₀, the next sequential FIFO.

4 Experimental Setup & Results

A number of experiments were conducted in order to investigate the feasibility of successfully utilizing the protocol and its implementation in a realtime cluster. We have implemented the endpoint and switch logic on two different Virtex-4 FX-100-FF1152 platforms. The HTG-V4-PCIE board from HitechGlobal is used for the endpoints and the ML423 from Xilinx is used as a switch. The HTG-V4-PCIE boards are connected to the ML423 board via SMA connectors. All the MGTs in the virtex-4 FX-100-FF1152 are brought out to SMA connectors on the ML423, making it an ideal platform for implementing the switch.

In our current implementation, the MGTs are configured to operate at a bit rate of 4Gbps resulting in a peak data rate of 400MBps in each direction. The MGT-to-logic interface is implemented to be 32-bit words, therefore, the receiver and transmitter logic operate at 100MHz. The application endpoint is also configured to operate at 100MHz. However, the receiver, transmitter, and application all operate in separate 100MHz clock domains.

4.1 Latency and Jitter

For the first experiment, we connected two endpoints, *A* and *B*, to the switch and measured the latency and the jitter of our implementation using a *ping-pong* test. In the ping-pong test, endpoint *A* sends a frame across the switch to endpoint *B*. Endpoint *B* sends the received frame back to endpoint *A*, through the switch, after *B* finishes receiving the frame from *A*. This test is repeated 40,000,000 times and the minimum, and maximum time taken to complete one round-trip is recorded over all of the iterations.

Table 1: Roundtrip Time (100 MHz Cycles).

Payload Size	Min	Max
1	122	126
2	124	128
4	128	132
8	136	140
16	152	156
32	184	188
64	248	252
128	376	380
256	632	636
512	1144	1148

Table 1 shows the results from the ping-pong test. The payload of the frame includes the error-checking word but excludes the source-routing word, SoF and EoF control words. The maximum round-trip jitter is five clock cycles and is the result of crossing clock domains. The total number of clock cycles required to transmit an *n*-word payload is given by equation (1) below:

$$T = 3 + n + T_o, \quad (1)$$

where *n* represents the clock cycles required to transmit *n* payload words and *T_o* is the overhead. The 3 in the equation represents one cycle each required to transmit the source-routing word, the SoF and EoF. Applying equation (1) to the minimum entry in each row of Table 1 shows that the minimum roundtrip overhead is 114 clock cycles (i.e., one-way overhead is 57 cycles).

When there are no frame collisions, the switch has a latency of 11 clock cycles beginning at the receipt of SoF by the receiver and the transmission of the SoF by the transmitter. This latency is the result of the two stage switching pipeline, one cycle selection, delays internal to the FIFOs, and one cycle each for writing to and reading from the FIFOs.

For the second experiment, we connected four endpoints, *A*, *B*, *C*, and *D* to the switch and measured the latency and jitter

using a four-point-traversal test. In the traversal test, the source-routing word is set to include the following eight hops: hops one and two from endpoint A to endpoint B via the switch; hops three and four from B to C; hops five and six from C to D; and hops seven and eight from D to A, completing the traversal. The switch decodes and updates the source-routing word for each pair of hops. We repeated the traversal test 40,000,000 times and recorded the minimum, and maximum time taken to complete each round-trip over all of the iterations.

Table 2: Roundtrip Time (100 MHz Cycles).

Payload Size	Min	Max
1	244	252
2	249	257
4	257	265
8	272	280
16	304	312
32	368	376
64	496	504
128	753	761
256	1264	1272
512	2291	2298

Table 2 shows the results from the four-point-traversal test. The frame payload includes the error-checking word and excludes the source-routing word, SoF and EoF. The maximum round trip jitter is nine cycles (or average time ± 4 clock cycles). The results show that the minimum round trip overhead is 228 cycles (*i.e.* four times the one-way overhead of 57 cycles). This is consistent with the observations in table 1.

4.2 Clock Synchronization

For the third experiment, we conducted a clock synchronization test between the switch and the endpoints. A distributed global clock is maintained at the switch and at all of the endpoints in the form of 64-bit counters that increment every clock cycle (at a frequency of 100MHz). The switch is responsible for synchronizing the global clock, and therefore, broadcasts its counter value to all endpoints periodically. At the time appointed by the schedule, each transmitter on the switch, retrieves the global clock counter value and transmits this value in a special frame indicted by a special SoF* control word. This frame consists of three words, the SoF*, and the two data words carrying the clock value (there is no CRC and EoF associated with this frame). Upon receiving the global clock from the switch, each receiver immediately updates its own global clock counter (instead of incrementing

it). Note that the clock value does not traverse any FIFOs in this process.

At the receiver, the clock is updated using the expression in equation (2):

$$C_{t+1} = C_t + ((R + \lambda - C_t) / 2), \quad (2)$$

where C_t is the current counter value, R is the received counter value, and λ is computed as follows:

$$\lambda = 3 + 19, \quad (3)$$

where λ is the expected one-way latency for transferring the synchronization frame. In (3), 3 represents the total size, in words, of the global clock synchronization frame. The 19 represents the maximum travel time for the clock frame (from the transmitter at the switch to the receiver at the endpoint). The one-way latency computed by equation (3) corresponds to a single switch network with eight connected endpoints. In a multi-switch network, where one or more switches are connected to the broadcasting switch or in succession with the switch, global clock synchronization can become complex.

The additional switches and overall network design can increase latency and introduce additional overheads. Furthermore, our global broadcast procedure may not be the most optimal synchronization method for a multi-switch network. There are other global clock synchronization methods (e.g. NTP and IEEE1588) that use multiple synchronization packages and a three-way handshake to update the global clock [10]-[13]. However, in a single switch FPGA network with predictable communication and synchronization stages due to a predefined schedule, a single frame one-way broadcast is suitable because we can easily compute the one-way latency using equations (1) and (3). For multi-switch networks, additional experimentation with our broadcast synchronization method will be required in order to characterize λ .

Our experiment resulted in a zero jitter in the expected global clock value computed using equation (2) at update frequencies 1Hz, 25Hz, 50Hz, and 100Hz. This is an improvement in the clock jitter over our previous implementation [1] in which we achieved a global clock jitter of 3 and 13 cycles for updates at 100Hz and 1Hz, respectively. Our zero jitter observation is a result of having eliminated the FIFOs in clock value update paths. At the switch, the global clock runs in a clock domain independent of all the transmitters. We implement a FIFO-free mechanism using registers to cross the clock value from the global clock's domain to each of the transmitters' domains (the register

essentially holds the global clock value for several cycles just as the scheduler is indicating to the transmitters that it is time to broadcast the global clock value – this enables the transmitter to read a stable global clock value).

5 Conclusions

The results reported in this paper demonstrate the improvements in performance of our implementation of the real-time link-level communication protocol as compared with our previous design and implementation [1]. Our prototype and experiments now report results that include a switch and exhibit a significant reduction in jitter of the distributed global clock. Our implementation supports a switch, with up to eight endpoint FPGAs.

In future work we plan to optimize our global clock synchronization method for a multi-switch FPGA network. We are also investigating the design and implementation of advanced schedulers for computational and communication activities in the network of FPGAs.

6 Acknowledgements

This work was supported in part by the following National Science Foundation grants: EPS-0903787, EPS-1006983, DUE-0513057.

7 References

- [1] R. D. Anderson and Y. S. Dandass, "A Protocol for Realtime Communication for FPGA Clusters using Multigigabit Transceivers," *22nd International Conference on Parallel and Distributed Computing and Communication Systems (PDCCS 2009)*, Louisville, KY, USA, September 24-26, 2009
- [2] Xilinx, *Virtex-4 RocketIO Multi-Gigabit Transceiver User Guide UG076 v4.1*, November 2, 2008, http://www.xilinx.com/support/documentation/user_guides/ug076.pdf, accessed March 29, 2011.
- [3] Xilinx, *Virtex-4 FPGA User Guide UG070 v2.6*, December 1, 2008, http://www.xilinx.com/support/documentation/user_guides/ug070.pdf, accessed March 29, 2011.
- [4] Xilinx, *Virtex-5 FPGA User Guide UG190 v5.3*, May 17, 2010, http://www.xilinx.com/support/documentation/user_guides/ug190.pdf, accessed March 29, 2011.
- [5] Altera, *Stratix IV Device Family Overview SIV51001-2.4*, June 2009, http://www.altera.com/literature/hb/stratix-iv/stx4_siv51001.pdf, accessed March 29, 2011.
- [6] Xilinx, *Aurora 8B/10B v3.1 for Virtex-4 FX FPGA, DS128*, April 24, 2009, http://www.xilinx.com/support/documentation/ip_documentation/virtex_4fx_aurora_8b10b_ds128.pdf, accessed March 29, 2011.
- [7] H. Kristian, O. Berge, and P. Häfliger, "High-Speed Serial AER on FPGA," in *Proceedings of the 2007 IEEE International Symposium on Circuits and Systems (ISCAS 2007)*, New Orleans, LA, USA, May 27-30, 2007.
- [8] M. Liu, W. Kuehn, Z. Lu., A. Jantsch, S. Yang, T. Perez, and Z. Liu, "Hardware/Software Co-design of a General-Purpose Computation Platform in Particle Physics," in *Proceedings of the 2007 International Conference on Field-Programmable Technology (ICFPT 2007)*, Kitakyushu, Kyushu, Japan, 2007.
- [9] A. X. Widmer and P. A. Franaszek, "A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code," *IBM Journal of Research and Development*, 27(5), 1983.
- [10] IEEE, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," IEEE Std 1588-2008, URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4579757>
- [11] J. Wu and J. Zhang and Y. Ma and M. Xie, "A Low-jitter Distributed Synchronous Clock Using DAC," *16th IEEE-NPSS Real Time Conference (RT' 09)*, Beijing, China, 2009
- [12] Y. Kang and J. Wu and M. Xie and Z. Yu, "A New Design for Precision Clock Synchronization Based on FPGA," *16th IEEE-NPSS Real Time Conference (RT' 09)*, Beijing, China, 2009
- [13] M. Zhang and S. Shen and Jian Shi and T. Zhang, "Simple Clock Synchronization for Distributed Real-Time Systems," *IEEE International Conference on Industrial Technology (ICIT 2008)*, Chengdu, China, 2008

Communicator Sensitive Static Analysis of MPI Collective Communication

Zhaofei Wang

National Laboratory for Parallel and Distributed Processing,
National University of Defense Technology,
Changsha, Hunan, P.R. China

Abstract— *Collective communication is widely used in MPI programs. However, its misuse may cause synchronization errors. This paper first proposes an extension to an existing static barrier analysis approach, so that it can check one necessary condition for correct collective communication. Since previous analyzers do not distinguish different communicators, they may report false alarms. This paper further presents a communicator sensitive collective communication analyzer. Moreover, this paper reports the results of comparative experiments on several real MPI programs. Compared with existing static analyzers, the proposed tool generates less false alarms, can check more communication behaviour, and is applicable to more programs.*

Keywords: collective communication, communicator, static analysis, MPI, synchronization

1. Introduction

MPI(Message Passing Interface) is an important parallel programming paradigm for high performance computing[1]. Collective communication is widely used in MPI programs to exchange information among a group of processes. MPI standard requires all members in each process group invoke the *same* sequence of collective subroutines for that group during whole program execution. Otherwise, some processes in a group may be stalled forever. This paper addresses how to detect this kind of synchronization errors.

MPI provides some communicator management subroutines which support process group partition. In addition, MPI allows textually unaligned collective subroutine invocations which are controlled by the so-called multi-valued expressions(MVEs). An MVE may evaluate differently for different processes, and can fork concurrent execution paths if used as a branching condition. Generally speaking, different processes may take different execution paths, and collective communication may involve any process group. Hence, it is challenging to develop precise collective communication analyzers.

Both dynamic and static tools have been developed to check collective communication in MPI programs. The former operate at run time and may miss errors that depend on specific program inputs[2], [3], [4], while the latter work at compile time and can check all potential program

```

01: MPI_Comm app_com, server_com;
02: void sub() {
03:     int am_server, num_ids, my_id, data;
04:
05:     MPI_Comm_size(MPI_COMM_WORLD, &num_ids);
06:     MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
07:
08:     if (my_id < (num_ids - 1)) {
09:         am_server = 0; //application processes
10:         //join subgroup app_com
11:         MPI_Comm_split(MPI_COMM_WORLD,0,my_id,
12:             &app_com);
13:         MPI_Bcast(data,1,MPI_INT,num_ids-1,
14:             MPI_COMM_WORLD);
15:     }
16:     else {
17:         am_server = 1; //the server process
18:         //join subgroup server_com
19:         MPI_Comm_split(MPI_COMM_WORLD,1,0,
20:             &server_com);
21:         data = 22;
22:         MPI_Bcast(data,1,MPI_INT,num_ids-1,
23:             MPI_COMM_WORLD);
24:     }
25:
26:     if (!am_server)
27:         //synchronize subgroup app_com
28:         MPI_Barrier(app_com);
29: }

```

Fig. 1: MPI program example

behaviour[5], [6], [7]. This paper concentrates on static tools. [6] presents a static barrier matching framework for MPI programs. In order to check more collective communication, we extend the approach in [6] to take into account other collective subroutines besides MPI_Barrier(). Nevertheless, the extended framework only checks one necessary condition for correct collective communication. Namely, all processes of each group invoke the same *number* of collective subroutines for that group.

The barrier analysis in [6] does not distinguish different communicators, possibly reporting false alarms. We use the MPI program in Figure 1 for illustration. MPI_Comm_split() is used to derive two communicators from MPI_COMM_WORLD, of which *server_com* specifies the group containing the process with ID (num_ids-1),

and *app_com* the group containing all other processes. Due to lines 9 and 17, *!am_server* at line 26 is multi-valued with respect to `MPI_COMM_WORLD`. However, it evaluates uniformly for the process group specified by *app_com*. The barrier at line 28 synchronizes the process group associated with *app_com*, which contains exactly the processes that will encounter the barrier. Hence the program will terminate normally. However, existing analyzers report a synchronization error because they regard *!am_server* as an MVE with respect to *any* communicator.

To reduce the false alarms mentioned above, it is necessary to distinguish different communicators. The candidate values of MVEs may be correlated with the invocations to communicator management subroutines. As shown in Figure 1, line 9 specifies 0 as a candidate value for the MVE *am_server* at line 26, while line 11 has an invocation to `MPI_Comm_split()`. Each process executes *either* all the two lines *or* none of them. Consequently, *am_server* evaluates to 0 at line 26 for each process of the group specified by *app_com*. By capturing this and other kinds of correlation, this paper presents a communicator sensitive collective communication analysis approach.

In summary, this paper makes the following contributions:

- 1) We propose a static collective communication analysis approach based on an existing static barrier analysis framework.
- 2) We present a communicator sensitive collective communication analysis approach.
- 3) By comparative experiments, we show the proposed analyzer is more precise, can check more communication behaviour and is applicable to more programs compared with previous tools.

2. Preliminaries

Communicators in MPI. An MPI communicator specifies a unique communication universe for a group of processes. For example, the predefined `MPI_COMM_WORLD` at line 5 of Figure 1 specifies the process group containing all processes.

Collective subroutines in MPI. MPI provides many subroutines for collective communication, such as `MPI_Barrier()` and `MPI_Bcast()`. The communicator parameter of each collective subroutine specifies the process group involved in the collective communication.

Textually unaligned collective subroutine invocations. If multiple textually different subroutine invocations specify the same collective communication, they are called textually unaligned. For example, the broadcasts at lines 13 and 22 in Figure 1 are textually unaligned.

Process identity. MPI provides `MPI_Comm_rank(MPI_Comm, int*)` to obtain the unique identities of processes. For example, after the execution of line 6 in Figure 1, *my_id* will hold the identity of the

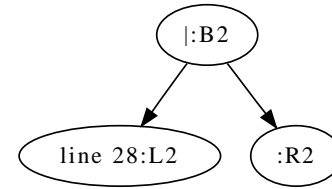


Fig. 2: Barrier tree for the program in Figure 1

calling process in the group specified by the communicator *com*.

Multi-valued expression(MVE) versus Single-valued expression(SVE). Derived from process identities, an MVE is an expression that may evaluate nonuniformly for a group of processes. On the contrary, an SVE evaluates identically among a group of processes. For example, in Figure 1, the condition at line 8 is an MVE derived from the process identity *my_id*, while *num_ids* at line 5 is an SVE which holds the number of all processes. Process identities are called MVE seeds.

Communicator management subroutine. MPI provides several subroutines for creating new communicators based on existing ones. After an invocation to `MPI_Comm_split(oldCom, color, newId, &newCom)`, the calling process of the group specified by the communicator *oldCom* will also belong to the new group specified by *newCom*. The new group will contain all calling processes that provide the same *color*. For example, *app_com* at line 12 is derived from `MPI_COMM_WORLD` in Figure 1.

3. Previous static barrier analysis

Given an MPI program, there are three stages in the static barrier analysis of [6].

- 1) A set of barrier trees are built, which compactly represents the barrier synchronization of the program.
- 2) Beginning with some MVE seeds, all possible MVEs in the program are tracked based on a set of inference rules which describe how new MVEs can be derived from old ones via data dependence.
- 3) Barriers are matched on barrier trees using MVE information. Counter examples will be generated if synchronization errors are detected.

In stage 1), a barrier tree is built for each function that calls `MPI_Barrier()` either directly or indirectly. Each barrier tree can have leaf nodes and non-leaf nodes. A leaf node may be labeled with an invocation to `MPI_Barrier()`. A leaf node may be labeled with the name of a function too. Such a function is called by the function associated with the barrier tree, and may invoke barriers. It may also have no label, denoting no barriers are involved. Each non-leaf node may be labeled with a `.`, `|` or `*`, corresponding to sequential composition, ordinary conditional branching and loop conditional branching respectively. Figure 2 shows the

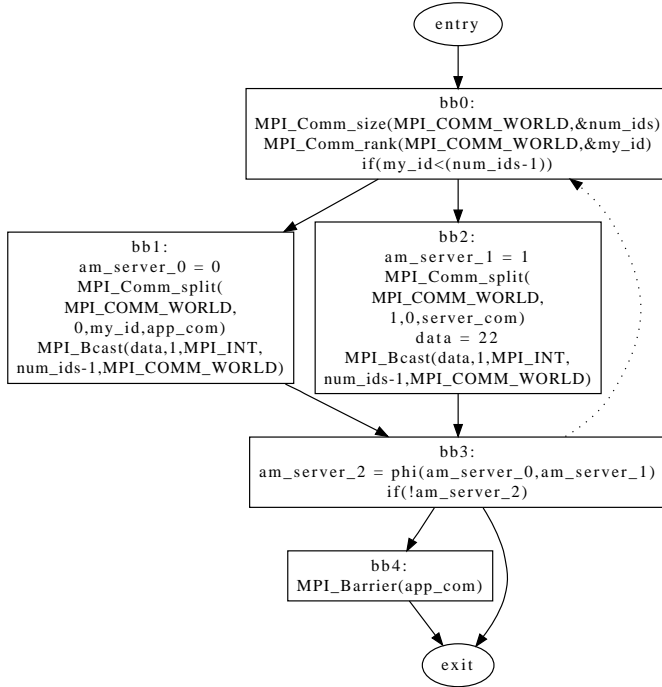


Fig. 3: The CFG for the Gated SSA form of the program in Figure 1

barrier tree for the MPI program in Figure 1. Each node of a barrier tree has a colon which separates its label on the left and its identifier on the right.

In stage 2), MVEs are tracked on Gated SSA program representation which facilitates data dependence analysis[8]. Figure 3 shows the control flow graph(CFG)[9] for the Gated SSA form of the MPI program in Figure 1. ϕ nodes are placed where multiple values for a variable merge. Each dotted edge connects a ϕ node to its controlling predicate.

!am_server at line 26 in Figure 1 is inferred to be an MVE as follows. The condition at line 8 is an MVE that has direct data dependence on the MVE seed my_id. As shown in Figure 3, the result of ϕ node am_server_2=phi(am_server_0,am_server_1) depends on the condition at line 8. Hence it is also an MVE. Since !am_server depends on the result of this ϕ node, the condition at line 26 is an MVE too. However, !am_server is regarded as an MVE with respect to *any* communicator due to communicator insensitive analysis.

In stage 3), each node in a barrier tree will be given a number that indicates how many barriers will be encountered if the subtree rooted at the node is executed. The notation \top is used to indicate any number of barriers may be encountered. Furthermore, let * be either a natural number or \top , it holds that $\top + * = \top$.

As shown in Table 1, the numbers of barriers for the nodes L2 and R2 are 1 and 0 respectively. Since the node B2 cor-

Table 1: Result of counting barriers in Figure 2

node	# of barriers
R2	0
L2	1
B2	\top

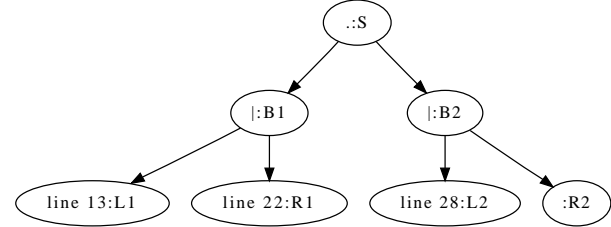


Fig. 4: Collective communication tree for the program in Figure 1

responds to a multi-valued branching condition and its two children have *different* number of barriers, a synchronization error will be reported. In fact, B2 corresponds to an MVE *only* with respect to MPI_COMM_WORLD. Moreover, the execution of the nodes L2 and R2 will both encounter *no* barriers with respect to MPI_COMM_WORLD. Hence, the program in Figure 1 can terminate normally and the reported error is a false alarm.

4. static collective communication analysis

Correct barrier usage does not amount to correct collective communication. As an extension to barrier tree, *collective communication tree* is introduced to compactly represent collective communication in MPI programs. We care for the following collective operations and their variants: MPI_Barrier(), MPI_Bcast(), MPI_Reduce(), MPI_Scatter(), MPI_Gather() MPI_Alltoall() and MPI_Scan(). The leaf nodes of a collective communication tree may denote these collective operations, while those of a barrier tree can only denote barrier invocations. Figure 4 shows the collective communication tree for the program in Figure 1.

As an extension to barrier matching, *collective communication matching* is introduced to check whether collective subroutines are used correctly. Nevertheless, only the following necessary condition for correct collective communication is checked: *all members of each process group should invoke the same number of collective subroutines for that group during whole execution.*

Similar to the calculation of barrier numbers, for each node n of a collective communication tree, we count the number of collective subroutine invocations that may be encountered when the subtree rooted at n is executed. If n denotes a multi-valued branching condition, and its two children have different numbers of collective subroutine

Table 2: Comparison of two approaches to counting the number of collective subroutine invocations

node	# of collective subroutine invocations	
	communicator insensitive	communicator sensitive
R2	0	0
L2	1	g(L2,MPI_COMM_WORLD)=0 g(L2,app_com)=1 g(L2,server_com)=0
B2	T	g(B2,MPI_COMM_WORLD)=0 g(B2,app_com)=T g(B2,server_com)=0
R1	1	g(R1,MPI_COMM_WORLD)=1 g(R1,app_com)=0 g(R1,server_com)=0
L1	1	g(L1,MPI_COMM_WORLD)=1 g(L1,app_com)=0 g(L1,server_com)=0
B1	1	g(B1,MPI_COMM_WORLD)=1 g(B1,app_com)=0 g(B1,server_com)=0
S	T	g(S,MPI_COMM_WORLD)=1 g(S,app_com)=T g(S,server_com)=0

invocations, then a synchronization error will be reported. Column 2 of Table 2 shows the number of collective subroutine invocations for each node in Figure 4.

5. Communicator sensitive collective communication analysis

To be communicator sensitive, MVE tracking should know whether an expression is multi-valued with respect to a given communicator. Furthermore, collective communication matching should calculate the number of collective subroutine invocations with respect to a given communicator.

5.1 Communicator sensitive MVE analysis

Given the result of a communicator insensitive MVE analysis, our analysis infers which MVEs may *actually* be SVEs with respect to certain communicators. Let C be the set of communicators, $c \in C$, E be the set of expressions, $e \in E$. A partial function f is introduced to describe communicator sensitive MVEs.

Definition 1: $f: E \times C \mapsto \{1,0\}$.

$$f(e,c) = \begin{cases} 1, & e \text{ is an MVE with respect to } c \\ 0, & e \text{ is an SVE with respect to } c \end{cases}$$

For convenience, $f(e,*)=y$ means $\forall c (c \in C \Rightarrow f(e,c)=y)$.

There are two kinds of correlation between the invocations to communicator management subroutines and the candidate values of MVEs in Figure 1.

- 1) Each process *either* executes both an `MPI_Comm_split()` and a definition of a candidate value for an MVE, *or* executes none of them. Hence, the MVE is actually an SVE with respect to the new communicator. lines 9 and 11 illustrate this.

```

01: ! node is an MVE seed
02: if (node .ge. nc*nc) then
03:   active_0 = .false.
04:   color_0 = 1
05: else
06:   active_1 = .true.
07:   color_1 = 0
08: end if
09: active_2 = phi(active_0, active_1)
10: color_2 = phi(color_0, color_1)
11: call mpi_comm_split(MPI_COMM_WORLD,color_2,
12:                    node,comm_setup,error)

```

Fig. 5: An MPI Fortran code segment derived from BT of NPB

- 2) For each multi-valued branching condition bc that controls the execution of an `MPI_Comm_split()` invocation, bc is actually an SVE with respect to the new communicator. For example, the MVE ($my_id < (num_ids - 1)$) evaluates identically to *true* with respect to `app_com` because line 8 controls the execution of line 11.

Derived from BT of NPB[16], the code segment in Figure 5 shows another kind of correlation as follows. `comm_setup` is a communicator derived from `MPI_COMM_WORLD`. `active_2` is an MVE with respect to `MPI_COMM_WORLD`, `active_0` and `active_1` are its two candidate values. However, `active_2` evaluates *identically* to one single candidate value with respect to `comm_setup`. The key of communicator sensitive MVE analysis is to capture these kinds of correlation mentioned above.

In compiler literature, control dependence graph(CDG) is used to describe the control dependence among program statements[9]. Figure 6 shows the CDG for the program in Figure 1. There is a path from node a to node b if and only if b is control dependent on a. *start* is a special CFG node that branches to the nodes *entry* and *exit*. R0 is one so-called region node that groups together the nodes having the same control dependence. T and F indicate a branching condition evaluates to true and false respectively.

Let $c_1, c_2 \in C$, $e_1, e_2 \in E$, S be the set of program statements, $s, s_1, s_2 \in S$, several relations are introduced as follows.

Definition 2: $CE \subseteq S \times S$.

Let s_1 and s_2 belong to the basic blocks bb_1 and bb_2 respectively. $(s_1, s_2) \in CE \iff (bb_1 = bb_2) \vee (bb_1 \text{ and } bb_2 \text{ are siblings in the CDG})$. $(s_1, s_2) \in CE$ means *either* s_1 and s_2 are both to be executed *or* none of them is. Here CE stands for control flow equivalence. For example, (line 9, line 11) $\in CE$ and (line 6, line 26) $\in CE$ in Figure 1.

Definition 3: $CV \subseteq E \times E$.

$(e_1, e_2) \in CV$ means e_1 is a candidate value of the MVE e_2 . For example, (color_0, color_2) $\in CV$ in Figure 5.

Definition 4: $CD \subseteq S \times E$.

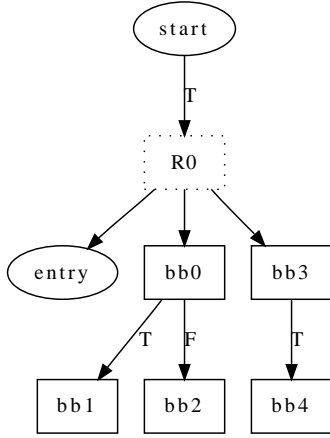


Fig. 6: Control dependence graph for the CFG in Figure 3

Let s and e belong to the basic blocks bb_s and bb_e respectively. $(s,e) \in CD \iff (bb_s \text{ is control dependent on } bb_e) \wedge (e \text{ is a branching condition})$. For example, (line 28, !am_server) $\in CD$ in Figure 1.

Definition 5: $\Phi \subseteq S$.

$s \in \Phi$ means s is a ϕ node. For example, (line 9) $\in \Phi$ in Figure 5.

Definition 6: $LHS \subseteq E \times S$.

$(e,s) \in LHS$ means e is the left hand side of s . For example, (am_server, line 9) $\in LHS$ in Figure 1.

Definition 7: $USE \subseteq E \times E$.

$(e_1, e_2) \in USE$ means e_2 is one operand of e_1 . For example, (!am_server, am_server) $\in USE$.

Let cms be `MPI_Comm_split(old_c, color, new_id, &new_c)`. Figure 7 shows the inference rules for communicator sensitive MVE analysis. In each rule, the formula above the line specifies the premise, while that below denotes inferred facts. Rules (1), (2) and (3) capture the three kinds of correlation mentioned above respectively. Rule (4) describes the propagation of communicator sensitive MVEs. Figure 8 provides 4 instances of MVE inference using the rules in Figure 7 respectively. The first, second and fourth inference instances are for the program in Figure 1, while the third one for that in Figure 5. Figure 9 shows the intraprocedural communicator sensitive MVE analysis algorithm. `en_queue()` appends an item to a queue, while `de_queue()` removes the head item of a queue. Table 3 shows the critical steps of the algorithm execution with the program in Figure 1 as the input.

5.2 Communicator sensitive collective communication matching

The key of communicator sensitive collective communication matching is to distinguish collective subroutine invocations with different communicator arguments. Let TN be the set of nodes in a collective communication tree, N be

$$\frac{(e_1, s) \in LHS \wedge (e_1, e_2) \in CV \wedge (s, cms) \in CE \wedge f(e_1, new_c) = 0 \wedge f(e_2, new_c) = 1}{f(e_2, new_c) = 0} \quad (1)$$

$$\frac{(cms, e) \in CD \wedge f(e, new_c) = 1}{f(e, new_c) = 0} \quad (2)$$

$$\frac{s \in \Phi \wedge (r, s) \in LHS \wedge t \in \Phi \wedge (color, t) \in LHS \wedge (s, cms) \in CE \wedge \forall v \in E((v, r) \in CV \Rightarrow f(v, new_c) = 0) \wedge \forall v \in E((v, color) \in CV \Rightarrow f(v, new_c) = 0) \wedge f(r, new_c) = 1 \wedge f(color, new_c) = 1}{f(r, new_c) = 0 \wedge f(color, new_c) = 0} \quad (3)$$

$$\frac{mve \in E \wedge f(mve, c) = 0 \wedge (e, mve) \in USE \wedge f(e, c) = 1}{f(e, c) = 0} \quad (4)$$

Fig. 7: Inference rules for communicator sensitive MVE analysis

Table 3: Critical execution steps of the algorithm in Figure 9 with the program in Figure 1 as input

step	inference	rule
1	(line 11, line 8) \Rightarrow $f(\text{my_id} < (\text{num_ids} - 1), \text{app_com}) = 0$	2
2	(line 9, line 11) \Rightarrow $f(\text{am_server}_2, \text{app_com}) = 0$	1
3	(line 19, line 8) \Rightarrow $f(\text{my_id} < (\text{num_ids} - 1), \text{server_com}) = 0$	2
4	(line 17, line 19) \Rightarrow $f(\text{am_server}_2, \text{server_com}) = 0$	1
5	line 26 \Rightarrow $f(!\text{am_server}_2, \text{app_com}) = 0$	4
6	line 26 \Rightarrow $f(!\text{am_server}_2, \text{server_com}) = 0$	4

the set of natural numbers, $n \in TN$ and $num \in N$. A function g is introduced as follows.

Definition 8: $g: TN \times C \rightarrow N \cup \{T\}$.

$g(n, c) = num$ means the execution of the subtree rooted at n will encounter num collective subroutine invocations that specify c as the communicator argument, while $g(n, c) = T$ means the number of such invocations can be arbitrary.

Figure 10 shows the rules for communicator sensitive calculation of the number of collective subroutine invocations. n_l and n_r denote the left and right child of n respectively. Rules (1), (2) and (6) are for leaf nodes, while rules from (3) to (5) are for non-leaf nodes. Using the program in Figure 1 as an example, Table 2 compares communicator insensitive and sensitive approaches to the counting of collective operations.

6. Implementation and evaluation

The proposed approach is implemented based on version 2.8 of LLVM compiler[10]. Combined with dragonegg[11], GCC can produce LLVM IR(intermediate representation)

$$\begin{array}{l}
(am_server_0, am_server_0 = 0) \in LHS \wedge \\
\quad (line\ 9, line\ 11) \in CE \wedge \\
\quad f(am_server_0, app_com) = 0 \wedge \\
(am_server_0, am_server_2) \in CV \wedge \\
\quad f(am_server_2, app_com) = 1 \\
\hline
\quad f(am_server_2, app_com) = 0 \\
(line\ 11, (my_id < (num_ids - 1))) \in CD \wedge \\
\quad f((my_id < (num_ids - 1)), app_com) = 1 \\
\hline
\quad f((my_id < (num_ids - 1)), app_com) = 0 \\
(active_2, line\ 9) \in LHS \wedge \\
\quad (color_2, line\ 10) \in LHS \wedge \\
\quad (line\ 9, line\ 11) \in CE \wedge \\
\quad (active_0, active_2) \in CV \wedge \\
\quad (active_1, active_2) \in CV \wedge \\
\quad f(active_0, comm_setup) = 0 \wedge \\
\quad f(active_1, comm_setup) = 0 \wedge \\
\quad (color_0, color_2) \in CV \wedge \\
\quad (color_1, color_2) \in CV \wedge \\
\quad f(color_0, comm_setup) = 0 \wedge \\
\quad f(color_1, comm_setup) = 0 \wedge \\
\quad f(active_2, comm_setup) = 1 \wedge \\
\quad f(color_2, comm_setup) = 1 \\
\hline
\quad (f(active_2, comm_setup) = 0 \wedge \\
\quad f(color_2, comm_setup) = 0) \\
\quad f(am_server_2, app_com) = 0 \wedge \\
\quad (!am_server_2, am_server_2) \in USE \wedge \\
\quad f(!am_server_2, app_com) = 1 \\
\hline
\quad f(!am_server_2, app_com) = 0
\end{array}
\tag{1}$$

$$\begin{array}{l}
(active_2, line\ 9) \in LHS \wedge \\
\quad (color_2, line\ 10) \in LHS \wedge \\
\quad (line\ 9, line\ 11) \in CE \wedge \\
\quad (active_0, active_2) \in CV \wedge \\
\quad (active_1, active_2) \in CV \wedge \\
\quad f(active_0, comm_setup) = 0 \wedge \\
\quad f(active_1, comm_setup) = 0 \wedge \\
\quad (color_0, color_2) \in CV \wedge \\
\quad (color_1, color_2) \in CV \wedge \\
\quad f(color_0, comm_setup) = 0 \wedge \\
\quad f(color_1, comm_setup) = 0 \wedge \\
\quad f(active_2, comm_setup) = 1 \wedge \\
\quad f(color_2, comm_setup) = 1 \\
\hline
\quad (f(active_2, comm_setup) = 0 \wedge \\
\quad f(color_2, comm_setup) = 0) \\
\quad f(am_server_2, app_com) = 0 \wedge \\
\quad (!am_server_2, am_server_2) \in USE \wedge \\
\quad f(!am_server_2, app_com) = 1 \\
\hline
\quad f(!am_server_2, app_com) = 0
\end{array}
\tag{2}$$

$$\begin{array}{l}
(active_2, line\ 9) \in LHS \wedge \\
\quad (color_2, line\ 10) \in LHS \wedge \\
\quad (line\ 9, line\ 11) \in CE \wedge \\
\quad (active_0, active_2) \in CV \wedge \\
\quad (active_1, active_2) \in CV \wedge \\
\quad f(active_0, comm_setup) = 0 \wedge \\
\quad f(active_1, comm_setup) = 0 \wedge \\
\quad (color_0, color_2) \in CV \wedge \\
\quad (color_1, color_2) \in CV \wedge \\
\quad f(color_0, comm_setup) = 0 \wedge \\
\quad f(color_1, comm_setup) = 0 \wedge \\
\quad f(active_2, comm_setup) = 1 \wedge \\
\quad f(color_2, comm_setup) = 1 \\
\hline
\quad (f(active_2, comm_setup) = 0 \wedge \\
\quad f(color_2, comm_setup) = 0) \\
\quad f(am_server_2, app_com) = 0 \wedge \\
\quad (!am_server_2, am_server_2) \in USE \wedge \\
\quad f(!am_server_2, app_com) = 1 \\
\hline
\quad f(!am_server_2, app_com) = 0
\end{array}
\tag{3}$$

$$\begin{array}{l}
(active_2, line\ 9) \in LHS \wedge \\
\quad (color_2, line\ 10) \in LHS \wedge \\
\quad (line\ 9, line\ 11) \in CE \wedge \\
\quad (active_0, active_2) \in CV \wedge \\
\quad (active_1, active_2) \in CV \wedge \\
\quad f(active_0, comm_setup) = 0 \wedge \\
\quad f(active_1, comm_setup) = 0 \wedge \\
\quad (color_0, color_2) \in CV \wedge \\
\quad (color_1, color_2) \in CV \wedge \\
\quad f(color_0, comm_setup) = 0 \wedge \\
\quad f(color_1, comm_setup) = 0 \wedge \\
\quad f(active_2, comm_setup) = 1 \wedge \\
\quad f(color_2, comm_setup) = 1 \\
\hline
\quad (f(active_2, comm_setup) = 0 \wedge \\
\quad f(color_2, comm_setup) = 0) \\
\quad f(am_server_2, app_com) = 0 \wedge \\
\quad (!am_server_2, am_server_2) \in USE \wedge \\
\quad f(!am_server_2, app_com) = 1 \\
\hline
\quad f(!am_server_2, app_com) = 0
\end{array}
\tag{4}$$

Fig. 8: Examples of MVE inference using the rules in Figure 7 respectively

from MPI programs based on C and Fortran[12]. We implemented the collective communication analysis as a pass in LLVM compiler which is divided into the following four phases.

- 1) collective communication tree construction.
- 2) communicator insensitive MVE analysis.
- 3) communicator sensitive MVE analysis.
- 4) communicator sensitive collective communication matching.

The first and second phases are essentially the reimplementation of the algorithms in [6], while the technical contributions of this paper lie in phases 3) and 4).

We conducted a comparative evaluation of our tool against PTP 4.0 and TASS 1.0. PTP(Parallel Tools Platform)[13] and TASS(Toolkit for Accurate Scientific Software)[7] are two suites of tools supporting static collective communication analysis. Table 4 shows the information about the chosen MPI programs. add2 and tsp are two applications using ADLB which is a library for load balancing[14]. DC is

Require: F is a function to be analyzed \wedge
f is as specified in Def. 1 \wedge
 $\forall e, c (e \in E \wedge c \in C \Rightarrow f(e, c) \text{ is undefined}) \wedge$
MVE_SET is the MVEs computed by communicator insensitive analysis

Ensure: f holds communicator sensitive MVEs in F

```

for all mve  $\in$  MVE_SET do
    f(mve, *) := 1
end for

CMS := all invocations to MPI_Comm_split() in F
for all cms  $\in$  CMS do
    while Rule (2) can be applied to cms do
        apply rule (2) to cms and change f accordingly
    end while
    CES := {s | s  $\in$  S  $\wedge$  (s, cms)  $\in$  CE}
    for all ces  $\in$  CES do
        if Rules (1) or (3) can be applied to (cms, ces) then
            apply the rule to (cms, ces) and change f accordingly
        end if
    end for
end for

work_queue := {(e, c) | e  $\in$  E  $\wedge$  c  $\in$  C  $\wedge$  f(e, c) = 0}
while work_queue  $\neq$   $\Phi$  do
    work_item := de_queue(work_queue)
    if Rule (4) can be applied to work_item then
        apply rule (4) to work_item and change f accordingly
        for all changes such as f(e1, c) := 0 do
            en_queue(work_queue, (e1, c))
        end for
    end if
end while

```

Fig. 9: Intraprocedural communicator sensitive MVE analysis algorithm

the document classification program in section 9.4 of a textbook[15]. BT and SP are from version 2.3 of NPB(NAS Parallel Benchmarks)[16]. LU-MZ is from multi-zone versions of NPB 3.3.1[16].

Table 5 shows the evaluation results. The \times means TASS can not parse the six programs due to its weak frontend. The ! means PTP would report a false alarm if it treats broadcasts in the same way as barriers. The * means PTP can not conduct barrier analysis for the programs written in Fortran. In fact, PTP would report a false alarm even if it could check BT due to communicator insensitive analysis. Table 5 shows straightforward adaptation of PTP to collective communication analysis would totally results in 6 false alarms, while our analyzer reports no spurious errors.

$$\frac{n \text{ has no labels}}{\forall c \in C(g(n, c) = 0)} \quad (1)$$

$$\frac{n \text{ is labeled with an collective subroutine invocation whose communicator argument is } c}{g(n, c) = 1} \quad (2)$$

$$\frac{n \text{ is labeled with } \cdot}{\forall c \in C(g(n, c) = g(n_l, c) + g(n_r, c))} \quad (3)$$

$$\frac{n \text{ is labeled with } |}{\forall c \in C\left(g(n, c) = \begin{cases} g(n_l, c) & g(n_l, c) = g(n_r, c) \\ \top & \text{otherwise} \end{cases}\right)} \quad (4)$$

$$\frac{n \text{ is labeled with } *}{\forall c \in C(g(n, c) = \top)} \quad (5)$$

$$\frac{n \text{ is labeled with the name of function } f \wedge f' \text{'s collective communication tree is } T_f \wedge n_f \text{ is the root node of } T_f}{\forall c \in C(g(n, c) = g(n_f, c))} \quad (6)$$

Fig. 10: Rules for calculating the number of collective subroutine invocations

Table 4: Information about the MPI programs used for evaluation

program	# of collective operations	# of communicators	language
add2	2	5	C
tsp	3	5	C
DC	2	3	C
BT	8	4	Fortran
SP	8	4	Fortran
LU-MZ	19	2	Fortran

Table 5: Results of comparative evaluation

program	# of false alarms		
	PTP	TASS	ours
add2	1	×	0
tsp	1 ¹	×	0
DC	1 ¹	×	0
BT	1*	×	0
SP	1*	×	0
LU-MZ	1*	×	0

7. Related work

Dynamic approaches have been proposed to check MPI collective communication[2], [3], [4]. [2] and [3] can check more necessary conditions for correct collective communication than our analyzer. For example, all members of a process group should provide consistent arguments for each common collective operation. Although practical, these methods may miss the defects triggered by specific program inputs.

TASS combines model checking and symbolic execution

to verify MPI programs against some safety properties including absence of deadlocks[7]. Although it can check all possible behaviour of program models, users have to specify such bounds as the number of processes. This not only means some burden on users, but also implies that *only* the behaviour within the specified bounds is checked.

[5] presents an inference system to statically detect barrier synchronization errors in Split-C programs. However, users are required to annotate the effects of procedures, and Split-C has no concept of communicator.

8. Conclusion

This paper proposes a communicator sensitive collective communication analysis approach based on an existing static barrier analysis framework. Experimental evaluation shows the presented analyzer is more precise, can check more communication behaviour, and is applicable to more programs compared with previous ones.

Acknowledgments

This work was supported by the National Natural Science Foundation of China under grant No.60725206. Wanwei Liu and Pei Fan helped me use Latex, and Xianjin Fu helped me submit papers. I would like to thank them.

References

- [1] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. International Journal of Supercomputer Applications, 8(3/4):165-414, 1994.
- [2] B. Krammer, K. Bidmon, M.S. Müller and M.M. Resch. Marmot: an MPI analysis and checking tool. In Proceedings of Parallel Computing, pp. 493-500, 2003.
- [3] C. Falzone, A. Chan, E. Lusk and W. Gropp. A portable method for finding user errors in the usage of MPI collective operations. International Journal of High Performance Computing Application, 21(2/4):155-165, 2007.
- [4] A. Vo, S. Vakkalanka, M. Delisi, G. Gopalakrishnan, R.M. Kirby and R. Thakur. Formal verification of practical MPI programs. In Proceedings of ACM Symposium on Principles and Practices of Parallel Programming, pp. 261-269, 2009.
- [5] A. Aiken and D. Gay. Barrier inference. In Proceedings of ACM Symposium on Principles of Programming Languages, pp. 342-354, 1998.
- [6] Y. Zhang and E. Duesterwald. Barrier matching for programs with textually unaligned barriers. In Proceedings of ACM Symposium on Principles and Practices of Parallel Programming, pp. 194-204, 2007.
- [7] S.F. Seigel et al. The toolkit for accurate scientific software. <http://vsl.cis.udel.edu/tass>, 2010.
- [8] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In Proceedings of International Conference on Supercomputing, pp. 414-423, 1995.
- [9] S.S. Muchnick. Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, 1997.
- [10] C. Lattner et al. LLVM compiler. <http://llvm.org>.
- [11] Dragonegg plugin for GCC. <http://dragonegg.llvm.org>.
- [12] GCC compiler. <http://gcc.gnu.org>.
- [13] Parallel Tools Platform. <http://www.eclipse.org/ptp>.
- [14] R. Lusk, S. Pieper, R. Butler and A. Chan. Asynchronous dynamic load balancing. <http://www.cs.mtsu.edu/rbutler/adlb>.
- [15] M.J. Quinn. Parallel programming in C with MPI and OpenMP. McGraw-Hill, New York, USA, 2004.
- [16] NAS Parallel Benchmarks. <http://www.nas.nasa.gov/NAS/NPB>.

SESSION

SIMULATION + NUMERICAL METHODS + PDE AND MATHEMATICAL PHYSICS AND ENGINEERING

Chair(s)

TBA

GPU Acceleration of Solving Parabolic Partial Differential Equations Using Difference Equations

David L. Foster

Electrical and Computer Engineering Department, Kettering University, Flint, MI, USA

Abstract- *Parabolic partial differential equations are often used to model systems involving heat transfer, acoustics, and electrostatics. The need for more complex models with increasing precision drives greater computational demands from processors. Since solving these types of equations is inherently parallel, GPU computing offers an attractive solution for drastically decreasing time to completion, power usage, and increasing the computation per dollar. However, since GPU computing involves a much different programming paradigm than traditional processors, techniques for optimizing solvers must still be developed. This paper presents several optimization strategies for accelerating solvers using CUDA to implement difference equations and compares their performances to a standard processor. The results demonstrate that different strategies should be used for different GPU cards, such as the C1060 and GTX 480, resulting in up to 197 times and 257 times single-precision and up to 133 and 163 times double-precision speedups respectively.*

Keywords: GPU programming, CUDA, parabolic partial differential equations, convection-diffusion-reaction equation

1 Introduction

Parabolic partial differential equations (PDEs) are useful in several problem spaces, such as heat transfer, acoustic modeling, and electrostatics. While numerical solution techniques are well-known, there are significant needs that can be addressed by solving them with GPU computing. With well-crafted algorithms, GPUs can potentially solve systems significantly faster, allowing decreased simulation times and/or increased resolution in the model. GPUs can also solve problems using an estimated one tenth to one twentieth of the power required by traditional supercomputing systems [1], thereby reducing costs.

This paper utilizes CUDA for GPU computing, which is an extension to several common programming languages that requires an NVIDIA-based video card for execution. NVIDIA GPUs are widely deployed and thus represents a very common computing platform. Additionally, NVIDIA's

Tesla series specifically targets high-performance computing. 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.

This paper proposes a set of optimizations for solving parabolic PDEs target several issues that arise when porting code to CUDA which must be deliberately addressed for efficient use of the GPU [2, 3]. First, GPU architectures generally mitigate memory latency by using large numbers of threads and extremely fast context switching instead of deep memory caches found in CPUs, although the Fermi architecture released in 2010 did add some memory caching. This requires optimizations to exploit enough parallelism to make sufficient threads available and to carefully manage the number of memory accesses required. Second, data accesses should be formed into coalesced reads and writes. See [2] for more details on coalesced accesses. Next, each SM has a limited amount of shared memory, usually 8 kB to 48 kB, that can be leveraged to reduce the number of RAM accesses. Finally, SMs are designed to execute a group of 32 threads, called a warp, concurrently. However, all threads in the warp must be executing the same instruction. Branching code creates divergence, which can drastically lower the throughput as only a subset of the warp executes. Carefully constructed code that limits divergence can minimize this effect.

Difference equations were used to solve several model parabolic PDEs, and optimizations were developed to address the above issues. Previous work in GPU computing focused on this topic include [4] which attained a 1.1 to 11 times performance improvement on two-dimensional parabolic PDEs using double-precision floating point arithmetic. Another effort targeted one-dimensional PDEs for market making real time pricing, and risk management achieved a 25 times speedup over a well-optimized CPU implementation using a single Tesla C1060, and a 38 times improvement using two C1060s by leveraging cyclic reduction [5]. A mixed precision method has been presented to solve ill-conditioned tridiagonal systems that previously were limited to CPU solutions with a 10-fold improvement[6].

Using the techniques proposed in this paper on two-dimensional parabolic PDEs, speed-ups of 197 times and 257 times were achieved using a C1020 Tesla and a GTX 480 respectively on single-precision floating point performance compared to an Intel i7 920. Speedups up to 133 and 163 times were attained on double-precision floating point performance.

This paper is organized as follows. Section 2 outlines the general approach used to solve parabolic PDEs using CUDA. Section 3 describes three optimization strategies that were implemented. Section 4 explains the experimental setup used. Section 5 presents the results of testing these optimizations, and Section 6 concludes the paper.

2 General Approach to Solver

This work uses difference equations to solve 2-dimensional parabolic PDEs of the form

$$u_t = u_{xx} + u_{yy} + A(x,y)u_x + B(x,y)u_y + C(x,y)u + f(x,y,t).$$

The boundary equations are expressed as

$$\begin{aligned} u(0,y,t) = g_1(y,t), u(1,y,t) = h_1(y,t) & \quad 0 < y < 1, t > 0 \\ u(x,0,t) = g_2(x,t), u(x,1,t) = h_2(x,t) & \quad 0 < x < 1, t > 0 \end{aligned}$$

and the system has the starting condition

$$u(x,y,0) = l(x,y)$$

where g_1 , g_2 , h_1 , h_2 , and l are determined by the system being modeled.

The two-dimensional grid is cast as a red-black array as shown in Figure 1. Red points are adjacent a black point on all four sides, and black points are adjacent to a red point of all four sides, much like a checkerboard. The red-black array is surrounded on all sides by a single row of points corresponding to the boundary conditions. The advantage to

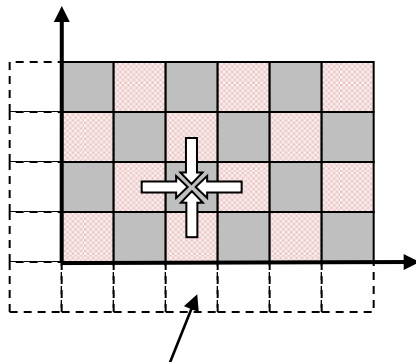


Figure 1 Red-Black Array showing an arbitrary black point's dependence on its four neighboring red points

this model is that all red points may be updated in parallel using values of neighboring black points. Then the black points can be updated using the new values of the red points.

The method to break this array into thread blocks needed to access memory efficiently and create enough thread blocks to occupy the GPU's SMs. This was accomplished by subdividing the array into sets of rows, such as 16 rows for example. Each block was assigned to $2N$ columns of a set in which each row contained N red points and N black points.

The general algorithm is represented by the following pseudo-code.

```

if using GPU: transfer red-black array to GPU RAM
for the required number of time steps
  update the boundary conditions
  update red points
  update black points
if using GPU: transfer red-black array to CPU RAM

```

3 Performance Optimizations

This section details the three main optimizations used.

3.1 Separated Red and Black Arrays

The first optimization split the unified red-black array into a red array and a separate black array as shown in the example in Figure 2. The separate arrays contained the same number of rows but had only half the number of columns as the original array. The separation was performed on the CPU, and the red and black arrays were passed to the CUDA kernel.

This optimization focused on two issues based on the following observation. When accessing N red points (similarly for black) in the unified array, they were interleaved with N black points. The accesses were not coalesced since the N required points were contained in a span of contiguous memory $2N$ points long. The default method would be to read in all $2N$ points and discard or save the black points for later use. With effective use of shared memory, this method can use all data read. However, writing the updated values of N red points back to the GPU card's RAM still required writing N points over a span of $2N$ contiguous locations with two writes. By separating the arrays, this write-back required only one write. Additionally, if a thread warp read in a set of interleaved red and black points from a unified array, the code must diverge so that the two subsets were handled differently. Using separated arrays avoided this divergence since the entire warp handled either red or black points.

3.2 Per Block Work Reduction

The second optimization slightly reduces the amount of work per thread block to reduce the number of sequential memory accesses. It was observed that to update N points,

4N+1 points must be read from memory: the N points being updated, the N points from the row above and from the row below, and N+1 points from the adjacent points on the same row. This requires one thread to make 5 sequential memory reads while the remaining threads in the block make only 4. To eliminate the extra latency, a block of N threads was coded to update N-1 points. This requires 4N-3 points from memory taking 4 sequential reads from each thread.

Using the separated arrays from Figure 2 with N=3, suppose a block of three threads is solving for all three red points in row 3. The block would make a request using all threads for (3,1), (3,3), and (3,5) from the red array, a second request for (4,1), (4,3), and (4,5) from black, a third request for (2,1), (2,3), and (2,5) from black, a fourth request for (3,0), (3,2), and (3,4) from black, and finally a fifth request for only (3,6) from black. If the block is solving for N-1 points, only the following four requests are needed. The block would make a request using all threads for (3,1), (3,3), and (3,5) from the red array and discard (3,5), a second request for (4,1), (4,3), and (4,5) from black discarding (4,5), a third request for (2,1), (2,3), and (2,5) from black discarding (2,5), a fourth request for (3,0), (3,2), and (3,4) from black.

Note that for a thread block of 128 threads handling 256 columns, the amount of work was reduced by only 0.78%. In many cases, this small decrease would be completely masked by using unutilized threads. For example, an array 1000 points wide is spanned by four thread blocks handling either 128-points or 127-points per block.

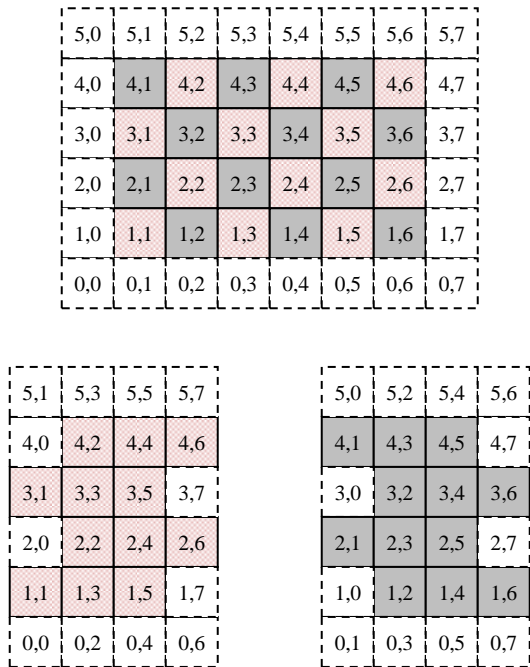


Figure 2 Dividing the Red-Black Array into separate red and black arrays. Indexes shown in the separate arrays are those from the original unified array.

3.3 Shared Memory

The final optimization reduced redundant memory reads. It can be easily seen that when updating a row of red points, for example, the black points on the same row and the black points in row below will be the upper and adjacent points respectively for updating the red points in the row below. Thus, these two rows of black points can be stored in the shared memory space, which may be faster than accessing RAM. Once the thread block loaded 4N points to update the first row of points, it only read an additional 2N points to calculate each additional row: N more red points and N more black points.

Using the separated arrays in Figure 2 as an example, suppose that a block of three threads first solved for the red points in row 4. The block would need to read row 4 from the red array and data from rows 3, 4, and 5 from the black array. If the block then solved for the red points in row 3, it would need data from row 3 of the red array and rows 2, 3, and 4 of the black array. If black rows 3 and 4 were saved in local shared memory, the block would only need to access RAM for the red row and row 2 of the black array.

4 Testing

This section details the four parabolic PDE models tested, the hardware setups, and the test parameters.

4.1 Model Equations Tested

The four models used were taken from [4] for purposes of comparison and since they have exact solutions for validation. All models are constrained by the conditions $0.0 \leq x \leq 1.0$, $0.0 \leq y \leq 1.0$, and $0.0 \leq t \leq 1.0$.

Model 1:

$$\begin{aligned}
 u_t &= u_{xx} + u_{yy} \\
 u(0,y,t) &= 0.0, u(1,y,t) = 0.0 & 0 < y < 1, t > 0 \\
 u(x,0,t) &= 0.0, u(x,1,t) = 0.0 & 0 < x < 1, t > 0 \\
 u(x,y,0) &= \sin(\pi x)\sin(\pi y) & 0 < x,y < 1
 \end{aligned}$$

This model has the exact solution

$$u(x,y,t) = e^{-2\pi^2 t} \sin(\pi x)\sin(\pi y).$$

Model 2:

$$\begin{aligned}
 u_t &= u_{xx} + u_{yy} - u_x - u_y - u \\
 u(0,y,t) &= e^{-t+y}, u(1,y,t) = e^{-t+1+y} & 0 < y < 1, t > 0 \\
 u(x,0,t) &= e^{-t+x}, u(x,1,t) = e^{-t+1+x} & 0 < x < 1, t > 0 \\
 u(x,y,0) &= e^{x+y} & 0 < x,y < 1
 \end{aligned}$$

This model has the exact solution

$$u(x,y,t) = e^{-t+x+y}.$$

Model 3:

$$\begin{aligned} u_t &= u_{xx} + u_{yy} + \sin(ax)\cos(ay)u_x - \cos(ax)\sin(ay)u_y - u \\ u(0,y,t) &= 0.0 \\ u(1,y,t) &= e^{(-2a^2-1)t} \sin(a) \sin(ay) \quad 0 < y < 1, t > 0 \\ u(x,0,t) &= 0.0 \\ u(x,1,t) &= e^{(-2a^2-1)t} \sin(a) \sin(ax) \quad 0 < x < 1, t > 0 \\ u(x,y,0) &= \sin(ax)\sin(ay) \quad 0 < x,y < 1 \end{aligned}$$

This model has the exact solution

$$u(x,y,t) = e^{(-2a^2-1)t} \sin(ax) \sin(ay)$$

Model 4:

$$\begin{aligned} u_t &= u_{xx} + u_{yy} + u_x + u_y + u + (1+xy)\cos(t) \\ &\quad - (1+x)(1+y)\sin(t) \\ u(0,y,t) &= \sin(t), u(1,y,t) = (1+y)\sin(t) \quad 0 < y < 1, t > 0 \\ u(x,0,t) &= \sin(t), u(x,1,t) = (1+x)\sin(t) \quad 0 < x < 1, t > 0 \\ u(x,y,0) &= \sin(\pi x)\sin(\pi y) \quad 0 < x,y < 1 \end{aligned}$$

This model has the exact solution

$$u(x,y,t) = (1+xy)\sin(t).$$

4.2 Hardware Setup

The computer used for testing had the following specifications:

- Intel Core i7 920 at 2.67 GHz
- Asus P6T Deluxe motherboard
- 6 GB of 1333 MHz DDR3 RAM
- EVGA GeForce 260 GTX video card
- 750 GB 7200 RPM hard drive
- Windows 7 Professional 64-bit
- CUDA version 3.1
-

The two graphics cards used for GPU computing were an NVIDIA Tesla C1060 with 4 GB of RAM and 240 cores in 30 SMs and an EVGA GeForce 480 GT with 1.5 GB of RAM and 480 cores in 15 SMs at stock clock speeds.

4.3 Software Parameters

Thread blocks were launched using 128 threads. This was the smallest value that allowed all 1024 thread slots available in the GPUs' SMs tested to be utilized. Larger values would have left more thread slots unoccupied if register and shared memory usage prevented 8 blocks from being assigned per SM.

Each of the models was tested using square arrays from 500 points per side to 4000 points per side in increments of 500 points. The number of rows contained in a set, as explained in Section II was tested at 16, 32, and 64 rows per set.

5 Results

The four versions of the kernel tested were CPU; GPU, which contained no optimizations; GPU-RB, which separated the data into separate red and black arrays; and GPU-RBS, which contained the separation of arrays, used the per-block work reduction, and used shared memory. All data refers to tests with 16-row pitch.

Several interesting patterns were noted in all tests. First, the throughput of the CPU implementation declined quickly in all tests from 500 to about 1500 points per dimension and gradually tapered off more as the data set sizes continued to increase. The rapid increase corresponded to the size of the data sets exceeding the L3 cache of the CPU, indicating that the CPU performance was likely limited by memory bandwidth. This effect can be seen in Table 1 for the C1060's performance on Model 2 for single-precision, showing the throughput of the different versions in millions of points updated per second based on the number of points per side.

Table 1 Tesla C1060 Throughput on Model 2 with Single-Precision Floating Point

Points per Side	CPU (Mpts/s)	GPU (Mpts/s)	GPU-RB (Mpts/s)	GPU-RBS (Mpts/s)
500	65.96	1078.39	1773.51	2754.44
1000	59.07	1230.28	1772.02	3182.40
1500	44.84	1266.84	1935.90	3770.77
2000	41.49	1302.58	1956.86	3854.73
2500	37.93	1170.87	1976.39	3940.19
3000	37.50	1120.09	1976.48	3958.37
3500	35.79	1149.27	2003.34	3983.28
4000	30.67	1152.60	1978.63	3989.22

Over the same range of 500 to 1500 points per side, the throughput of the GPU versions greatly increased. This effect was related to the number of threads that were created based on the dimensions of the problem. From the previous section, a GPU block of 128 threads processed 254 columns and either 16, 32, or 64 rows. For a 16-row block and a 500 by 500 point array, this yielded only 64 blocks. Since the C1060 can hold 240 blocks of this size concurrently, the smaller data sets did not fully occupy the card, and the MPs suffered more idle time during memory accesses. Larger data sets allowed the GPUs to mitigate this latency more efficiently. It should also be noted that in addition to fully occupying the card on smaller data sets, the 16-row pitch also showed slightly better throughput by a few percent once over 32- and 64-row pitches.

Omitting the small data sets that didn't result in full occupancy of the GPU, Table 2 and Table 3 show the mean,

minimum, and maximum speeds for both GPUs on the models for 1500 to 4000 points per dimension. Full test results are not shown for space considerations. For both cards, the separated red and black arrays yielded a substantial speed increase over the basic GPU implementation with a unified red-black array. As can be seen by comparing the GPU and GPU-RB columns, this optimization often yielded speeds of around 50%.

For the Tesla, which is based on the GT200 series GPU that does not have memory caching, the use of shared memory gave significant benefits. For single-precision, the additional optimizations in GPU-RBS gave a 27.9% to 99% increase over the GPU-RB performance. For double-precision, it yielded almost double the performance for Models 1 and 2, significant increases in Model 4, and made notably little improvement in Model 3. However, since the 400-series GPUs, like that in the GTX 480, do have a memory cache, the use of shared memory required too much overhead to produce a benefit. Thus, the performance over GPU-RBS compared just GPU-RB was often about 10% slower, with around a 25% decrease for Model 2 with single precision.

It is also noteworthy that the newer GTX 480 did not dominate the C1060 as might be expected. The C1060 is based on the same GPU as the 280 GTX video cards, which were the predecessors to the GTX 480. In single-precision performance, the C1060 was comparable to the GTX 480 on Model 1, dominated on Model 2, was within 90% of the performance on Model 3, and was faster on data sets 3000 points per side and smaller on Model 4. For double-precision, the C1060 dominated the GTX 480 on Models 2 and 4, but the GTX 480 dominated on the other two. Therefore, these results show that different optimizations are beneficial based on the GPU architecture used, and that older architectures may still be more advantageous for certain problems.

6 Conclusions

This research demonstrates several advantageous techniques for accelerating difference equation solvers for two-dimensional parabolic PDEs. An important result is that optimization strategies differ based on the underlying GPU hardware, and effective GPU computing programming practices need to account for this.

7 Acknowledgments

This research was supported by an equipment donation from the NVIDIA Corporation as part of the Academic Partnership Program.

Table 2 Mean, Minimum, and Maximum Speedups on 1500 by 1500 to 4000 by 4000 point data sets on single-precision floating point numbers

	GPU	GPU-RB	GPU-RBS
Model 1 C1060	31.82, 28.13, 38.09	52.15, 42.75, 64.49	102.97, 82.33, 129.41
Model 1 GTX 480	63.85, 53.27, 79.47	101.23, 82.70, 129.27	92.77, 73.20, 119.47
Model 2 C1060	31.68, 28.25, 37.58	52.61, 43.17, 64.51	104.63, 84.09, 130.07
Model 2 GTX 480	53.15, 47.13, 64.41	74.88, 65.60, 89.47	56.81, 49.03, 68.61
Model 3 C1060	49.44, 46.44, 51.80	80.88, 70.29, 91.07	103.45, 90.74, 91.07
Model 3 GTX 480	83.50, 77.19, 95.07	110.37, 93.70, 124.70	102.68, 88.10, 115.54
Model 4 C1060	80.40, 74.83, 84.75	132.53, 108.24, 146.92	177.22, 148.76, 197.27
Model 4 GTX 480	126.66, 100.82, 164.79	189.86, 146.82, 257.16	154.45, 116.43, 216.70

Table 3 Mean, Minimum, and Maximum Speedups on 1500 by 1500 to 4000 by 4000 point data sets on double-precision floating point numbers

	GPU	GPU-RB	GPU-RBS
Model 1 C1060	26.05, 19.33, 31.85	45.04, 32.46, 55.55	89.48, 63.72, 111.64
Model 1 GTX 480	71.06, 48.97, 90.16	132.27, 91.70, 163.52	123.37, 84.71, 158.43
Model 2 C1060	26.10, 20.09, 31.60	45.32, 33.53, 55.70	90.46, 66.36, 112.43
Model 2 GTX 480	49.41, 37.96, 61.84	75.94, 55.13, 95.96	71.41, 51.21, 90.39
Model 3 C1060	19.15, 16.60, 21.15	27.11, 25.04, 28.90	27.83, 25.58, 30.04
Model 3 GTX 480	59.54, 53.72, 64.95	68.38, 60.35, 75.55	67.16, 59.44, 74.59
Model 4 C1060	39.33, 33.70, 44.01	68.90, 60.84, 75.44	117.27, 104.70, 133.84
Model 4 GTX 480	61.13, 53.58, 74.56	87.41, 76.18, 105.32	81.16, 70.65, 98.73

References

- [1] NVIDIA. (2010, Nov. 23). *Tesla C2050/C2070 GPU Computing Processor Overview*.
- [2] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: a Hands-On Approach*: Morgan Kaufmann Publishers, 2010.
- [3] J. Sanders and E. Kandrot, *CUDA by Example, An Introduction to General-Purpose GPU Programming*: Addison Wesley, 2010.
- [4] C.-W. Hsieh, *et al.*, "Rapid Performance of Parabolic Problems using Convection Diffusion Reaction on GPU Accelerator," presented at the PDPDA'10, Las Vegas, NV, USA, 2010.
- [5] D. Egloff, "High Performance Finite Difference PDE Solvers on GPUs," QuantAlea GmbH, Zurich, Switzerland 2010 2010.
- [6] D. Goddeke and R. Strzodka, "Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid," *IEEE Transactions on Parallel and Distributed Systems* vol. 22, pp. 22-32, Jan. 2011 2011.

Lock-Graph: A Tree-Based Locking Method for Parallel Collision Handling with Diverse Particle Populations

Mark Lewis¹ and Cameron Swords¹

¹Computer Science, Trinity University, San Antonio, TX, USA

Abstract—*This paper builds on earlier work that used a spatial grid for locking to provide physically accurate parallel collision handling. Instead of using a grid, this work uses a spatial tree. The tree is better able to handle heterogeneous particle populations. The method was specifically developed to handle a granular flow impact simulation where one large body impacts a population of smaller bodies. The large size of the impactor leads to a breakdown in the grid based locking strategy because grid cells are uniformly sized and must be large enough to enclose the largest particle in the population plus the relative velocity distribution multiplied by a time step. The tree allows the regions that get locked to scale in size based on local characteristics, making it possible to handle dramatic size differences. Unfortunately, it also has more overhead than the grid so using it when it is not needed can slow simulations down.*

Keywords: Simulation, collisions, parallel, discrete-event

1. Introduction

Parallelizing collisions in a physically accurate way is inherently challenging. This is because, unlike longer range forces such as gravity, collisions are temporally sensitive. The behavior is sensitive to the order of collisions, as one collision can alter the path of particles to prevent other collisions or to make them happen at different times. Fortunately, the short-range nature of collisions means that over a fixed period of time, one can set bounds on how far the effect of a particular collision will travel. This range can be thought of as the speed of sound in the medium times the length of time being considered. This logic leads to spatial locking where collisions are processed in order, but parallel processing of future collisions is allowed as long as those collisions are far enough away from one another (1; 2; 3).

This dependence on temporal ordering is not uncommon in discrete event simulations, indeed, it is the norm. Significant effort has gone into finding ways to parallelize such systems (4; 5; 6). A common solution to this is to implement the ability to roll back changes (7; 8; 9). For a general discrete event simulation that solution typically allows greater parallelism than trying to keep things together, but it comes at the expense of memory. For this application, roll back is not really an option. The storage of the rollback information would have a computational cost that would rival what was gained from parallelism. More importantly,

these simulations are often memory constrained. Particle counts in physical, collisional simulations can get as high as a few times 10^8 , and for some situations that is the limit which constrains what can be done. In the case of planetary rings, a simulation with 10^8 particles is near the minimum required for getting decent resolution with a ring that goes all the way around the planet. Even then it must be a narrow ring which precludes some types of work. Adding the memory overhead of storing older states for particles we can roll back to would further depress the maximum simulation size for a given cluster configuration.

In our previous work (1), locking was done using a uniform grid. The grid cells are made large enough that during one time step, the probability of a particle colliding with another particle whose center is located two or more cells away is effectively zero. This works well enough for the majority of ring simulations. Indeed, for simulations where the population is fairly homogeneous in size and spatial distribution and the velocity distribution has a small dispersion, this method is optimal. For those simulations, the same grid can be used as the spatial data structure used for searching potential collision pairs as well, so there is a net benefit in sharing the data structure.

Such a grid is less ideal for finding collisions when the particle distribution has heterogeneities. For example, if there are a small number of significantly larger particles, if the particles are very non-uniformly distributed in space, or if there are spatial variations in the velocity distribution. For these situations, a tree is better as the primary spatial data structure for finding collisions. We also found, for one particular simulation, that the grid approach to locking could be untenable.

Figure 1 shows a granular particle simulation in which a marble is dropped into a dish of silicon spheres. When this was first attempted using the grid for locking, the simulation bogged down to a point where it wasn't going to finish in a reasonable period of time. This prompted the development of the method described here.

2. Methodology

The key goal for the new method was to be flexible enough to handle the situation shown in fig. 1. Building off a spatial tree was a natural approach because these simulations use a tree instead of a grid for collision finding and we could

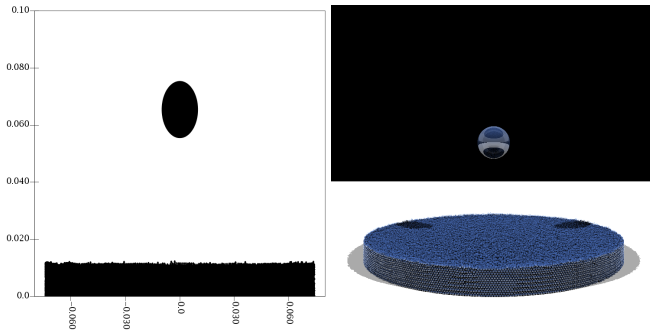


Fig. 1

THIS SHOWS A FRAME FROM THE SIMULATION THAT MOTIVATED THIS WORK. AN INITIAL SIMULATION WAS DONE IN WHICH UNIFORM SMALL PARTICLES WERE DROPPED AND ALLOWED TO SETTLE IN A DISH. THAT CONFIGURATION WORKED FINE WITH THE GRID LOCKING. THEN A SINGLE IMPACTOR WAS ADDED THAT WAS 20 TIMES LARGER IN RADIUS THAN THE OTHER PARTICLES AND DROP IT INTO THEM. THIS CAUSED THE GRID LOCKING TO FAIL, AS THE LARGE SIZE AND HIGH VELOCITY OF THE IMPACTOR LED TO A LARGE GRID SIZE.

reap the benefits of having a mutual data structure for both purposes. To be competitive with the grid, the method also needed to be fast. The grid allows $O(1)$ access to the eight adjacent cells to check if a collision is being processed in one of them so we can know if it is safe to process a collision in the current cell. Solutions that run through the tree would be $O(\log n)$, where n is the number of particles, and would have a higher coefficient due to the overhead of moving through the tree. The method presented here uses the tree as the spatial data structure, but it also builds a graph through the tree that we call the lock-graph, which retains the $O(1)$ access aspects of the grid and the dynamic and spatially variable nature of the tree.

The basic idea of this method is that in the spatial tree certain nodes are labeled as lock-nodes. Only the lock-nodes are significant in determining if a collision can be processed or not. The path from the root of the tree to any leaf (where particles are contained), can contain only a single lock-node. Each lock-node must be at least as large as the largest particle plus a multiple of the velocity dispersion for the particles under that node. The tree knows these values already as they are used in the collision pair searching algorithm. It would be simple enough to have a recursive function that runs through the tree finding the nodes that should be lock nodes based on the specified size criteria. However, we also need to know about “adjacent” nodes. For any given lock node there will be other nodes around it that are within range and must be searched for collisions. Simple recursion doesn’t give us that.

To make it so we can quickly find the adjacent nodes, we need to have a graph where edges connect the lock nodes

in the tree that are adjacent for the purposes of collision finding. This structure is what we call the lock graph. It contains all the lock nodes with edges between any lock nodes that contain particles that could impact one another. This structure can be built at the same time that lock nodes are picked using recursion over two arguments. This pseudo-code shows how it works.

```
def buildLockGraph(n1:Node,n2:Node) {
  if(n1==n2) {
    val lock = n1.numParts>=0 ||
      n1.size<SCALE*(n1.maxRad+n1.searchRadius)
    if(!lock1) {
      buildLockGraph(n1.firstChild,n1.firstChild)
      buildLockGraph(n1.firstChild,n1.secondChild)
      buildLockGraph(n1.secondChild,n1.secondChild)
    }
  } else {
    val dx = n1.mid.x-n2.mid.x
    val dy = n1.mid.y-n2.mid.y
    val dz = n1.mid.z-n2.mid.z
    val dist = sqrt(dx*dx+dy*dy+dz*dz)
    val searchDist = n1.searchRadius+n1.maxRad+
      n2.searchRadius+n2.maxRad
    if(dist-0.866*(n1.size+n2.size)<=searchDist) {
      val lock1 = n1.numParts>=0 ||
        n1.size<SCALE*(n1.maxRad+n1.searchRadius)
      val lock2 = n2.numParts>=0 ||
        n2.size<SCALE*(n2.maxRad+n2.searchRadius)
      if(lock1 && lock2) {
        addLockEdge(n1,n2)
      } else if(lock1) {
        buildLockGraph(n1,n2.firstChild)
        buildLockGraph(n1,n2.secondChild)
      } else if(lock2) {
        buildLockGraph(n2,n1.firstChild)
        buildLockGraph(n2,n1.secondChild)
      } else {
        buildLockGraph(n1.firstChild,n2.firstChild)
        buildLockGraph(n1.firstChild,n2.secondChild)
        buildLockGraph(n1.secondChild,n2.firstChild)
        buildLockGraph(n1.secondChild,n2.secondChild)
      }
    }
  }
}
```

There are two top level cases. The two nodes could be the same or they could be different. If they are the same, we only decide if the node should be a lock-node. If it isn’t, then we need to recurse three times to the different possible combinations of the two children of the node.

If the nodes aren’t the same, the distance between their center points is calculated and that is checked against their sizes and the search distance of the two nodes. If they are too far apart, the recursion terminates. Otherwise the code determines if either one is a lock node. If both are, that edge is added to the graph. If only one is, that one recurses against both children of the other. If neither is, all four combination of the children are recursed on.

This function is called using the root of the tree for both $n1$ and $n2$. When it is done, all lock nodes have been identified and all edges have been added. This form of two argument recursion can have many uses in spatial work when the objective is to find pairs of entities. One other application was presented in (10). It can also be nicely parallelized so that it does not add to sequential overhead in the application.

At this point the lock-graph can be used in much the

same way the grid was used previously. When the processing begins on a collision, the lock-node for each of the two particles is set to being locked. To make this efficient, a map from particles to nodes is made when the graph is built. When the collision is done, the lock is released. To check if a particular collision is safe to be processed, the code checks that node for each of the two particles in the collision and all the other nodes directly connected to them in the graph.

The check in the pseudocode for whether or not it is a lock node includes a factor called SCALE. This can be adjusted to move the lock-nodes up or down in the tree. Moving them up leads to fewer lock nodes that have more particles in them, but there are fewer connections between lock nodes. Moving them down has the opposite effect. Increasing the number of particles in a lock node can lead to over-locking if the lock-nodes get too big. However, the fact that the check for whether a collision is safe or not must run through all the edges out of a node means that lower connectivity can have a positive performance impact on one of the most common operations in the simulation. The impact of this is explored in the next section.

3. Analysis

The method was implemented in a collisional dynamics simulation code in C++. This code has been used previously for modeling of planetary rings (11; 12; 13). It was tested on two general types of systems, the system for which it was created shown in fig. 1 and a system using a small cell in a planetary ring shown in Figure 2. Both simulations included a bit under 200,000 particles and they were run through ten time steps to get the timing results. In addition, each system was run in two configurations called early and late. The early was the initial conditions and the late was after the system had evolved to an equilibrium. For the ring simulation the early is truly uniform with a dynamically cold particle distribution. The late was after two orbits when particle self-gravity and collisions had altered the distribution and particles were beginning to clump a bit. For the silicon grain simulations, the early stage had a cube of well spaced particles completely inside the cylinder falling down to the surface. The late simulation is what is shown with the particle at rest at the bottom of the cylinder.

To do the time testing we used a server with 4 Quad-Core AMD Opetron 8350 processors, running at 2.0GHz processor, giving a total of 16 cores. The C++ code base was compiled using the x86 Open64 compiler from AMD. The compile flag were “-Ofast -mso -march=auto -openmp”. All multithreading was done using OpenMP. The “-apo” autoparallelization flag for the compiler was not used as the objective is to test the parallelism coded explicitly into the simulation. It is worth noting that choice of compiler can be significant. While the x86 Open64 compiler was selected under the belief that it would out perform the GNU C++ compiler, incomplete results from that compiler show

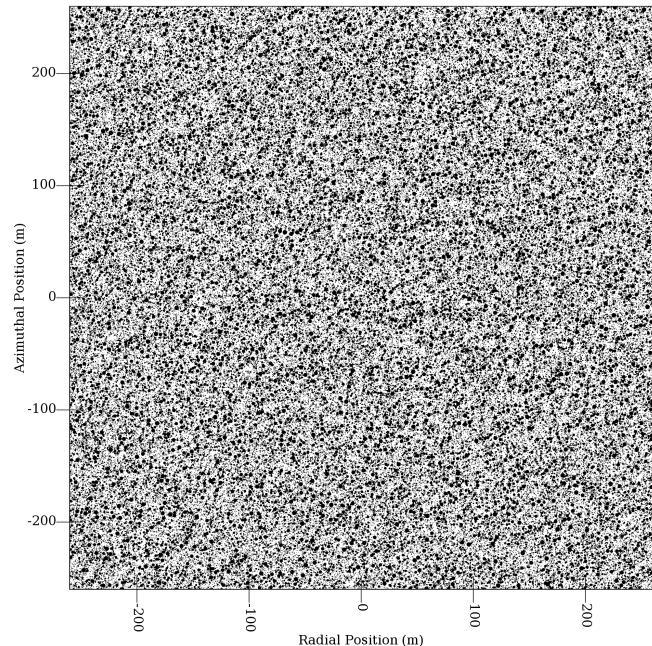


Fig. 2

THIS FIGURE SHOWS A FRAME FROM THE RING PARTICLE SIMULATION THAT WAS USED FOR TIMING RESULTS. THIS IS WITHOUT THE MOON. THE SIMULATIONS WITH A MOON HAD A 20-M PARTICLE PLACED AT THE ORIGIN.

that things aren't so clear cut. For some of the situations presented, the GNU compiler produces slightly faster code.

The results in the tables show what was reported by the Linux time command. Each simulation was run five times with the mean and standard deviation being presented.

The first table shows timing results for the ring simulation shown in fig. 2. This was the situation for which the grid was originally developed. The particles are fairly homogeneously spread out and the particle distribution is quite flat. The timing data here shows that for this system, larger lock-nodes are generally better and that a SCALE of ~6.0 is ideal. It isn't too surprising that even with that scale, the grid implementation is slightly better.

Table 2 shows the times for the ring simulation when a moonlet, 10 times larger in radius than the largest other particles, was dropped into the middle of the simulation. This setup was a bit artificial because the system was only advanced 10 steps which isn't enough time to let smaller particles settle on the surface of the moonlet, but it does test the graph with a heterogeneous particle size distribution. The results for this test were very odd as the graph timing for the initial conditions was highly variable and none of the graph runs performed as well as the grid.

A possible explanation for this is shown in Figure 3. The large particle forces one large lock node in the graph.

Scale	Graph Early (s)	Graph Late (s)
1.2	44.4 ± 0.4	58.3 ± 0.9
2.0	44.6 ± 0.1	57.0 ± 1.0
4.0	42.0 ± 0.1	47.7 ± 0.2
6.0	39.9 ± 0.2	43.7 ± 0.2
8.0	39.2 ± 0.1	43.8 ± 0.1

Table 1

THIS TABLE SHOWS TIMING RESULTS FROM RING SIMULATIONS WITH A POPULATION OF PARTICLES WHOSE SIZES WERE PULLED FROM A DIFFERENTIAL POWER-LAW DISTRIBUTION WITH A SLOPE OF -2.8 SPANNING A FACTOR OF TEN IN RADIUS. TIMES FOR THE GRID BASED LOCKING WITH THIS SIMULATION WERE 37.1 ± 0.1 EARLY AND 42.0 ± 0.1 LATE. FOR THE LARGER GRAPH SCALE THE GRAPH WAS COMPARABLE IN SPEED TO THE GRID, BUT NEVER SUPERIOR.

Scale	Graph Early (s)	Graph Late (s)
1.2	95.2 ± 5.7	75.3 ± 0.8
2.0	136.4 ± 6.2	75.5 ± 1.2
4.0	177.8 ± 7.2	67.5 ± 0.6
6.0	112.2 ± 3.0	66.2 ± 0.7
8.0	212.7 ± 7.6	66.5 ± 1.1

Table 2

THIS TABLE SHOWS TIMING RESULTS WHEN A MOONLET, 10 TIMES LARGER IN RADIUS THAN THE LARGEST BACKGROUND PARTICLES, WAS ADDED TO THE SIMULATION. THE TIMING FOR THE GRID IN THIS SITUATION WAS 93.0 ± 2.0 EARLY AND 57.9 ± 0.3 LATE.

That node has a very high connectivity. If there aren't enough collisions happening in nodes that aren't adjacent to that large node, the workload will be unbalanced and the processing will become more sequential. It is worth further exploration into whether having a more natural particle configuration or a larger cell would favor the graph more.

We also have time results for the granular flow simulations that prompted the development of the graph-lock method. This simulation differs from the ring simulations in another very significant way, the distribution of particles is distinctly 3-D. Planetary rings are remarkably flat. While the 3-D aspect of the rings is very significant to the dynamics (14), all the particles are close to the orbital plane. That is significant for the grid, which only breaks the simulation region up in 2-D. It would be possible to make a 3-D grid, but memory overhead quickly becomes a problem with 3-D grids that have decent spatial resolution, and while the neighbor count in 2-D is only eight, it goes up to 26 for 3-D, increasing the overhead of lock checking.

The results without the marble show how different this system is from the ring simulations. Here, increasing the scale of the lock nodes has little impact on performance when the particles are spread out early and slows it down when they are densely packed after having settled to the bottom. In addition, while the grid performs roughly the same speed as the graph for the early system, it is slightly

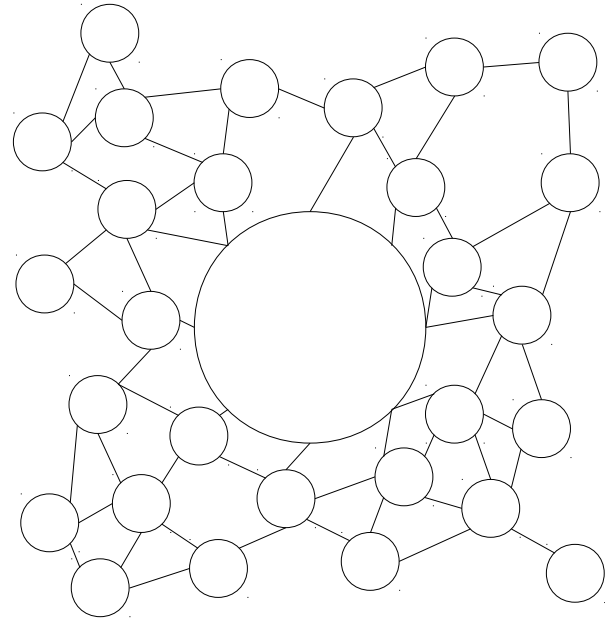


Fig. 3

THIS IS A PICTORIAL REPRESENTATION OF A LOCK GRAPH WHEN THERE IS A SINGLE LARGE PARTICLE. THE SIZE OF THE NODES ROUGHLY REPRESENTS THE AREA IT COVERS. THE NODE WITH THE LARGE PARTICLE HAS A MUCH HIGHER CONNECTIVITY AND IT IS POSSIBLE THAT CHECKS FOR ITS LOCK COULD ACTUALLY SLOW THINGS DOWN IF THERE AREN'T ENOUGH COLLISIONS HAPPENING AWAY FROM IT TO KEEP THE LOAD BALANCED.

Scale	Graph Early (s)	Graph Late (s)
1.2	21.0 ± 0.6	860 ± 39
2.0	20.6 ± 0.3	859 ± 30
4.0	20.7 ± 0.8	913 ± 40
6.0	20.5 ± 0.7	936 ± 22
8.0	19.3 ± 0.5	968 ± 33

Table 3

THESE ARE TIMING RESULTS FOR THE SILICON GRAIN SIMULATIONS WITHOUT THE DROPPING MARBLE. THE GRID BASED LOCK TIMING FOR THESE SIMULATIONS WERE 19.5 ± 0.5 EARLY AND 1004 ± 38 LATE.

slower later on. In all cases though, the 3σ bounds overlap.

Adding the marble is where the graph should in theory stand out. Again, the performance is fairly flat with graph scale except for the largest size which caused it to run significantly slower. The grid performed similarly to the largest scale for the early time. It is interesting to note that in this situation there isn't much of a difference between the early and late simulations using the graph. One result that seems a bit unusual is that the late setup runs faster with the marble than it did without. This implies that having the marble located up high actually changes the morphology of

Scale	Graph Early (s)	Graph Late (s)
1.2	647 ± 37	667 ± 50
2.0	668 ± 41	663 ± 39
4.0	686 ± 52	705 ± 24
6.0	667 ± 46	646 ± 13
8.0	2529 ± 1323	2780 ± 1185

Table 4

THESE ARE TIMING RESULTS FOR THE SILICON GRAIN SIMULATIONS WITH THE DROPPING MARBLE. THE GRID BASED LOCK TIMING FOR THESE SIMULATIONS WERE 2654 ± 1153 EARLY AND $\gg 150,000$ LATE.

the graph as a whole in such a way that it allows greater parallelism. Lastly, with the marble at the late time it wasn't possible for us to get good time bounds on the grid method. After a few days of running it still hadn't completed the first time step which allows us to give the limit in the table caption.

4. Conclusions

The lock-graph method of locking parallel threads adds another option to the toolkit of those working in spatially oriented discrete event simulations. While specifically designed for collisional simulations, it could be used to good effect in any discrete event application where events have a spatial distribution and exhibit locality. Unlike rollback methods, this does not require significant memory overhead and can be used for simulations that are often memory constrained. The use of two-argument recursion to build the lock-graph means that it has a worse case performance of $O(N \log N)$, the same was what would be required to build or maintain the tree data structure.

As is so often the case in simulation, the lock-graph is no silver bullet. There are trade-offs and some situations where it is not the ideal approach. For systems that are naturally flat and mostly homogeneous, a grid will work better. However, the use of a kD-tree for the lock-graph means that it will efficiently scale to higher dimensions where a grid would become infeasible and the ability to handle 3-D systems that exhibit significant heterogeneity makes it appropriate for systems where the grid-based approach breaks down.

Acknowledgments

This work was supported by grants from NASA Origins and NSF AAS.

References

- [1] M. Lewis and B. L. Massingill, "Multithreaded collision detection in java," in *PDPTA*, H. R. Arabnia, Ed. CSREA Press, 2006, pp. 583–592.
- [2] M. Lewis, M. Maly, and B. L. Massingill, "Hybrid parallelization of n-body simulations involving collisions and self-gravity," in *PDPTA*, H. R. Arabnia, Ed. CSREA Press, 2009, pp. 324–330.
- [3] B. L. Massingill and M. Lewis, "Parallelizing a collisional simulation framework with plpp (pattern language for parallel programming)," in *PDPTA*, H. R. Arabnia, Ed. CSREA Press, 2006, pp. 608–614.
- [4] R. M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, pp. 30–53, October 1990. [Online]. Available: <http://doi.acm.org/10.1145/84537.84545>
- [5] E. Mascarenhas, F. Knop, and V. Rego, "Parasol: a multithreaded system for parallel simulation based on mobile threads," in *Proceedings of the 27th conference on Winter simulation*, ser. WSC '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 690–697. [Online]. Available: <http://dx.doi.org/10.1145/224401.224711>
- [6] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, pp. 1–19, March 1995. [Online]. Available: <http://portal.acm.org/citation.cfm?id=201627.201628>
- [7] R. M. Fujimoto, K. S. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley, "Large-scale network simulation: How big? how fast?" in *MASCOTS*. IEEE Computer Society, 2003, pp. 116–.
- [8] R. M. Fujimoto, *Parallel and Distributed Simulation*. John Wiley & Sons, Inc., 2007. [Online]. Available: <http://dx.doi.org/10.1002/9780470172445.ch12>
- [9] A. M. Law, *Simulation Modeling and Analysis*, 4th ed. McGraw-Hill Higher Education, 2007.
- [10] M. Lewis and H. Levison, "A tree-based hamiltonian for fast symplectic integration," in *CSC*, H. R. Arabnia, Ed. CSREA Press, 2008, pp. 30–36.
- [11] M. C. Lewis and G. R. Stewart, "Expectations for Cassini observations of ring material with nearby moons," *Icarus*, vol. 178, pp. 124–143, Nov. 2005.
- [12] —, "Features around embedded moonlets in Saturn's rings: The role of self-gravity and particle size distributions," *Icarus*, vol. 199, pp. 387–412, Feb. 2009.
- [13] S. J. Robbins, G. R. Stewart, M. C. Lewis, J. E. Colwell, and M. Sremčević, "Estimating the masses of Saturn's A and B rings from high-optical depth N-body simulations and stellar occultations," *Icarus*, vol. 206, pp. 431–445, Apr. 2010.
- [14] H. Salo, "Numerical simulations of dense collisional systems," *Icarus*, vol. 90, pp. 254–270, Apr. 1991.
- [15] H. R. Arabnia, Ed., *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications, PDPTA 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 1*. CSREA Press, 2006.

Multi-agent System Simulation in Scala: An Evaluation of Actors for Parallel Simulation

Aaron B. Todd¹, Amara K. Keller², Mark C. Lewis² and Martin G. Kelly³

¹Department of Computer Science, Grinnell College, Grinnell, IA, USA

²Department of Computer Science, Trinity University, San Antonio, TX, USA

³Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA

Abstract—Multi-agent system (MAS) simulation, a growing field within artificial intelligence, requires the creation of high-performance, parallel, and user-friendly simulation frameworks. The standard approach is to use threads and shared memory. The drawbacks of this approach are the common concurrency pitfalls of race conditions and performance loss due to synchronization. Our goal was to evaluate the feasibility of an alternate model of concurrency, actors. An actor can be thought of as a very lightweight thread that does not share memory with other threads, instead communicating only through message passing. Actors seem to be a natural fit for this task, since agents are concurrently processed objects that communicate with each other through message passing. We write an actor framework and an equivalent threaded framework in the modern object-functional JVM language Scala and compare their performance. We conclude that the actor model seems like a natural fit, but its performance is inferior to that of the threaded model. Despite this drawback, it shows great promise due to its elegance and simplicity. When scaling to multiple machines, the advantages of actors will almost certainly outweigh any performance costs.

Keywords: MAS, Scala, Parallel Simulation, Actors, AI

1. Introduction and Background

An important simulation problem is that of multi-agent systems (MAS). A MAS “can be defined as a loosely coupled network of problem solvers that interact to solve problems that are beyond the individual capabilities or knowledge of each problem solver. These problem solvers, often called agents, are autonomous and can be heterogeneous in nature” [1]. Because the definition of an agent is so broad, MAS can model many different situations, such as economic, social, and political activity. Of course, creating realistic simulations is not easy and requires collaboration with economics and/or other social sciences. However, MAS provide the technological framework to make such modeling possible.

Implementations of MAS frameworks vary widely, but the core parallelization model behind many of them is that of threads and shared memory. Although this approach can work, it suffers from the hazards of concurrent data

modification and other race conditions. These issues make testing and debugging of the frameworks very difficult, and solutions to these problems typically incur performance penalties and programmer headaches.

Another approach to parallel programming is the actor model [2]. An actor can be defined as a lightweight process that communicates with other actors through message passing. These messages are buffered in the actors’ mailboxes for the actor to respond to. Actors do not share any memory with other actors, and they all process concurrently. Message passing is asynchronous. This model avoids the numerous shared memory pitfalls associated with the conventional threaded model of concurrency. The best known actor implementation is in the language Erlang, a functional language designed for efficient fault-tolerant distributed systems [3].

A MAS framework based on this actor model would avoid many of the concurrency issues afflicting the threaded frameworks. In such a framework, it would be natural for each agent to be an actor. Communication between agents would then be done by the actor’s message passing methods and the simulation would be implicitly parallel. While it would help eliminate many reliability issues and make frameworks much easier to write, this approach is only practical if the performance is comparable to that of the threaded frameworks. Our goal is to perform a performance evaluation of an actor model MAS framework written in the language Scala by comparing it to a comparable threaded framework.

Scala is a fairly new programming language that has been in development at EPFL in Switzerland since 2003. It uses an object-functional paradigm and compiles to the JVM, which allows seamless calls to Java libraries and code. It has static type checking and, as a result, takes full advantage of HotSpot JVM implementations and typically runs at the same speed as Java programs [4].

There has been significant work on the Scala language that is well documented in the field of programming languages. As a result, the language pulls in many of the best ideas from the field [5] [6] [7] [8]. The name Scala is short for Scalable Language, a property that makes it ideal for generating Domain Specific Languages (DSLs). This can be extremely beneficial in the field of simulation, where many simulation packages have basically built up their own

DSLs over the years [9]. The design of Scala allows libraries to be written such that they look and operate as normal language features. These features, among many others, were significant factors in motivating us to use the language for these frameworks. The Scala actor library is unusually high-performing relative to other JVM actor implementations. Together, Scala's extensibility and the convenience of a JVM platform makes Scala a great language choice [10]. This is the reason why we chose it over other actor-supporting languages such as Erlang.

2. Related Work

To our knowledge, we are one of the first groups to consider using Scala's actor library for parallel simulations. Some research has been done into the feasibility of using Erlang for MAS. The implementation described by Varela et al. does map Erlang's lightweight processes directly to agents, but in this case, each agent is a collection of these processes. Unfortunately, Varela's work does not provide a performance evaluation [11]. His group's primary motivations for using Erlang over Java was the much more natural fit they saw between Erlang and the coding of agent behavior, combined with Erlang's excellent support for distributed computing [?]. Performance appears to have been good enough that any weakness was compensated for by the convenience of using a language that fits the problem well.

3. Scala

Scala, as a language very closely related to Java, borrows much of its syntax; however, it omits semicolons and eliminates the need for extensive boilerplate code. The most noticeable difference is a type inference system for limiting type specification when such specification would be redundant. These tweaks make Scala much easier to read and feel more like a scripting language even though it actually runs on the robust JVM platform.

Listing 1: Scala Int to String class.

```
class Foo {
  def bar(arg: Int): String = arg.toString
}
```

Listing 2: Java Int to String class.

```
public class Foo {
  public String bar(Int arg) {
    return Integer.toString(arg);
  }
}
```

In addition to these simple variations, Scala has many more subtle differences and features, including very natural

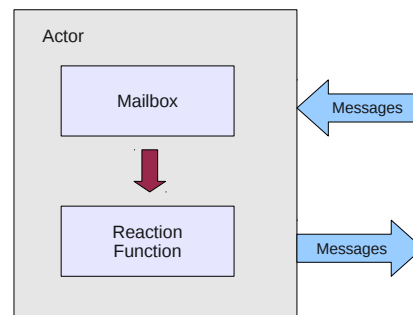


Fig. 1: An actor. Mailbox buffers messages until the reaction function processes them.

support for pattern matching. This is very useful because the primary action of an actor is to react to a message with an action that varies based on message content. Excellent pattern matching syntax makes this a painless process.

Scala can represent this reaction function as a number of case statements. These are matching functions evaluated in sequence on the input. These can match on type or value and can pull inner values by automatically applying extractors. Extractors are present in all standard library collections and can be automatically generated for a class by declaring it with the case keyword. In the following example, the input message is checked to see if it is an Int, the String "foo", or a Bar that contains 5. A default is provided, but since only one match can occur, it does not require special syntax.

Listing 3: A simple match block.

```
msg match {
  case msg: Int => println('An integer.')
  case 'foo' => println('Found foo.')
  case Bar(5) => println('The bar.')
  case _ =>
}
```

4. Actor Framework

4.1 Scala Actors

Scala has an excellent actor library. Since Scala is a JVM language, this library is quite complicated because the JVM is not designed to run actors efficiently [12] [13]. The library works around this limitation by leveraging the language's functional nature and by using exceptions to navigate the call stack. Event based actors, a variant that does not block an underlying thread while waiting for messages, are implemented as closures waiting to be called with an input message. Upon receiving a message, the actor's associated closure is then scheduled on an available executor thread. When an actor is finished reacting to the message, it throws a suspend actor exception, which returns the actor to its idle

state. By implementing actors in this way instead of mapping them directly to JVM threads, they become very lightweight. This gives them substantially better performance than many other JVM-based actor implementations [10].

A Scala actor can be defined by extending the Actor trait. All actor functionality is contained in the act method, which must be defined by the subclass. A simple actor is defined below. React is the method actors call to “react” to the next message in their mailbox. Due to its non-returning nature, a result of its exception-based implementation, a for or while loop will not work, so the library provides a special loop function. Inside react is a matching function similar to the one above. The “!” method is used to send messages to actors, and sender can be used to refer to the actor that sent the message being reacted to. This actor responds to Pings with Pongs and Pongs with Pings.

Listing 4: A Ping-Pong actor.

```
class PingPong extends Actor {
  def act() {
    loop {
      react {
        case Ping => sender ! Pong
        case Pong => sender ! Ping
      }
    }
  }
}
```

4.2 Agents

In our framework, each agent is a subclass of actor. This design allows the framework to inherit all of the actor message passing functionality, and since actors process concurrently, there are only a few agent features left to implement. The primary task is to convert the event based system of actors reacting to a system in which agents iterate through time steps. We do this by creating a clock actor that sends agents messages which trigger the processing of steps. Once each agent finishes its step, it sends a message back to the clock indicating that it is finished. Once the clock receives finished messages from each agent, it sends out new DoStep messages. The method that agents call upon receiving a DoStep message is called doStep.

4.3 Stepping Algorithm

This results in the following step algorithm for the framework. A runner object initializes the simulation and then the clock object loops a step function of the form:

- Send DoStep message to each agent.
- Wait until every agent has replied with an EndStep message.
- Cleanup and repeat if not finished.

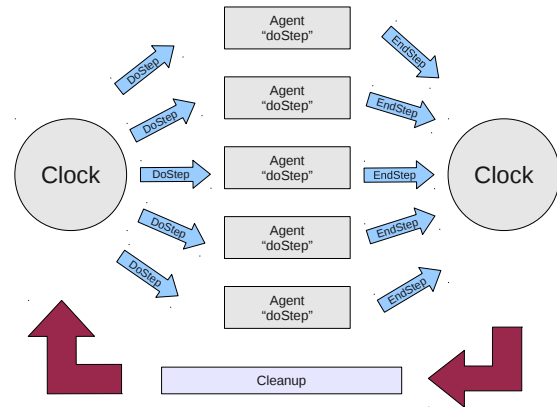


Fig. 2: Clock sends DoStep messages, Agents call doStep method, Agents inform clock that they are finished. Cleanup and repeat.

Inside the doStep call, agents perform any simulation logic they need to with the option of placing some code to handle messages in a dedicated handleMessage method. This was done because many messages in the simulation are simple information requests. Processing these requests inside doStep creates an unnecessary hassle because they interrupt the logical flow of the agent’s code. This external method also allows agents to respond to information requests once finished with their core logic without having to know how many requests they will receive. In the case in which receiving a message is a core part of the agent’s logic, they can still react to it in doStep.

These methods are defined in a core Agent abstract class that is extended by simulation writers as needed. The act method of an agent is a simple loop reacting to messages. The messages it receives are of two possible types. The first is FrameworkMessage. This type of message is used by the framework to tell the agent what it should be doing. Examples include messages to start the simulation and do steps. The second type is a SimulationMessage. This message is sent between agents as part of their steps. An example is an information request message. The react sends each type to its appropriate handleMessage method. In the case of a SimulationMessage, this is the handleMessage method defined by the simulation writer.

Upon receiving a DoStep message, the agent calls the doStep method written by the simulation writer. Inside this method, agents are free to send messages to other agents using the conventional agent message send method, “!”. When reacting to messages, a special react method specifically for agents must be used. This implementation provides a number of benefits. Using a number of Scala language features allows this method to return normally, unlike the usual react. It also allows an agent to use its

handleMessage method in addition to the partial function passed in to react to messages and does so in a way that is transparent to the simulation writer.

Listing 5: An example agent subclass.

```
class AnAgent(friend: AnAgent) extends Agent {
  def doStep() {
    friend ! Greet("Hello")
    agentReact {
      case Greet(msg) => println(msg)
    }
  }
}
```

4.4 Implementation

The implementation of this react is the most complex component of the actor framework. While it is not strictly necessary, its inclusion makes writing simulations substantially easier through the avoidance of bizarre control flow. For that reason, the performance hit from its overhead was deemed acceptable.

The actual implementation is as follows: When agentReact is called, the first action taken is to use Scala's delimited continuations library to store all remaining computation in the doStep method as a continuation. Then the agent reacts to incoming messages with both the partial function given to agentReact and the agent's coreReact partial function. If the supplied partial function is used to react to the message, the continuation is then called using the actor library method andThen, which takes a function to apply once react has finished. If the message is handled by the coreReact partial function, this react block is repeated, again by calling andThen. The primary drawback to this system is that all looping constructs with continuation-creating code in their bodies must be implemented with an understanding of continuations. Since Scala's constructs are not aware, a special agentReactWhile function is defined.

Scala's continuations library is one of the few that is not implemented purely as library code. It works by using a compiler plug-in to perform a transformation of all code contained in the delimited continuation to continuation-passing style (CPS). The result of this transformation is that instead of returning normally, all functions take an argument, which is the function to apply to the result. This function can then be saved and stored as a "continuation" instead of being evaluated immediately. The details of this transformation are available in [14].

One might question the use of the Scala continuations library to achieve this behavior instead of using andThen following react. While the latter probably has better performance, it would result in fairly convoluted code. If an agent had many reacts interspersed with pieces of computation, using andThen would require a deep nesting of function

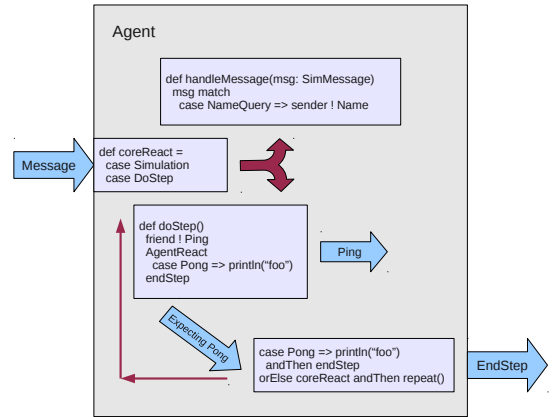


Fig. 3: Agent message processing. If SimulationMessage, send to handleMessage. If DoStep, perform agent behavior. If agentReact called, move to modified react which forwards message back to coreReact and then repeats agentReact by calling the saved continuation if not the expected message.

Fig. 4: Note how a normal react results in deep nesting.

<p>(a) Returning react.</p> <pre>def doStep() { agentReact { partialFunction } agentCode agentReact { partialFunction } agentCode }</pre>	<p>(b) Normal react.</p> <pre>def doStep() { agentReact { partialFunction } andThen { agentCode agentReact { partialFunction } andThen { agentCode } } }</pre>
---	--

calls. When writing more complex logic, this nesting would begin to make code unreadable. One way to think of our use of continuations is that the compiler's CPS transform is simply a way to get rid of these extra brackets in order to make the code more readable. In the original work on Scala actors, the authors also state that the lack of a return on react is due to the lack of first-class continuations and, had there been a technique similar to our implementation, they would have used it instead [12]. When continuations were added to Scala, one of the examples used was a modification to react that made it return, which was an implementation very similar to ours [14].

Messages in this framework are defined very simply. FrameworkMessage and SimulationMessage are both traits that extend a Message trait. Framework messages with parameters are defined as case classes, where case is a Scala keyword that automatically generates simple constructor methods and extractors for the pattern matching done

in react. If there are no parameters, singleton messages are used to save memory. Simulation writers define their message types by extending the `SimulationMessage` trait with additional case classes.

There is also a `CentralAgent` class that is given information about all agents in the simulation. This agent is useful because it allows normal agents to perform actions such as requesting a reference to a random agent very easily. It is always in a react loop waiting to respond to messages from agents. If they desire, simulation writers can easily extend it with additional functionality.

The final component of the simulation is the system by which agents are initially created. A simulation writer must provide an iterator that produces agents in sequence. Any initial setup, such as setting agent data, must be done by this iterator. A future task is to implement a DSL that simplifies this process.

4.5 Example Simulation

Our primary benchmark is a simulation of `CommunicatingAgents`. These agents are friends with every other agent, and each step they send a Hello message to each friend. Once they have received a response HelloBack message they finish their step. When `doStep` is called, the agent sends Hello to each friend and counts how many were sent. It then moves into an `agentReact` loop, in which it waits for a response from every friend. Concurrently with `agentReact`, `handleMessage` responds to Hello messages.

Listing 6: `CommunicatingAgent` implementation.

```
class CommunicatingAgent
  (val friends: ListBuffer[CommunicatingAgent])
  extends Agent {
  def doStep() {
    var count = 0
    for( a <- friends ) {
      a ! Hello
      count += 1
    }
    agentReactWhile(count > 0) {
      case HelloBack => count -= 1
    }
    endStep
  }
  def handleMessage(msg: SimulationMessage) {
    msg match {
      case Hello => sender ! HelloBack
    }
  }
}
```

5. Threaded Framework

In order to evaluate the performance of the actor parallelism approach to MAS, we also wrote simulations in a

framework using a threaded model. This framework provides an equivalent environment to write simulations such as ours using conventional concurrency constructs and a master-slave program structure. It was written as an initial exploration of the merits of Scala as a MAS simulation language. The step algorithm is as follows.

- Master calls “runStep” on slave.
- The slave iterates through its agent list.
- “doStep” is called for each agent.
 - Returns a list of messages the agent has sent.
 - Messages to agents placed in their mailboxes.
- Slave processes agents again
 - “handleMessages” is called for each agent.
- Repeats until there are no messages.
- Slave finishes; informs master.
- Cleanup and advance time step.

Unfortunately, this framework suffers from many of the problems associated with the threaded concurrency model. Race conditions between threads make testing and debugging difficult, and frequent synchronization of access to shared memory slowed program execution down.

6. Experiments and Results

6.1 Testing Goals and Methods

Our primary goal in this research was to determine how well Scala’s actor library performs relative to threads for MAS simulation. To do this, we run a number of benchmarks for our two frameworks that stress different framework components. For these tests we recorded wall clock time between simulation start and end, not counting agent creation. Our test machines contained dual Xeon 5450 quad-core CPUs at 3.0 GHz and 16 GB of memory. All tests were run with a maximum Java heap size of 14 GB and simulations were run for 20 steps.

6.2 Agent Number

Our first test was to see how execution time scaled while increasing the number of agents. To test this we wrote an agent class where each agent performs no actions and just immediately ends its step after starting. Here we had some very bizarre findings. The threaded framework scaled linearly with the addition of new agents, which is what was expected. The actor framework did not. As Fig. 6 shows, at 100k agents, the execution times become very random. Our plausible explanation for this behavior is the interaction between the actor library, the JVM, and multi-CPU machines. Since actors have special scheduling that leverages exceptions, the clock actor is frequently suspended and resumed. As this is happening, it could be switching between threads, cores, or the physical CPUs. If this were to thrash CPU cache or interfere with exception processing in the JVM, execution would be substantially slowed down

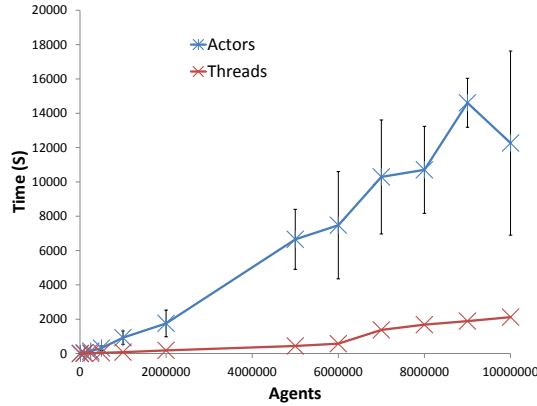


Fig. 5: Execution time scaling for simple agents.

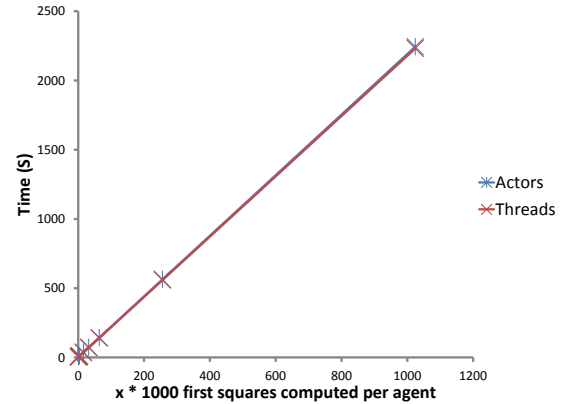


Fig. 7: Execution time scaling as agent computational workload varies.

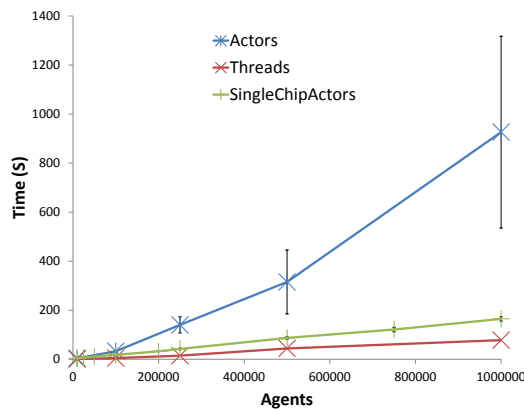


Fig. 6: Note very large uncertainty for dual CPU actors, and essentially none for single CPU actors.

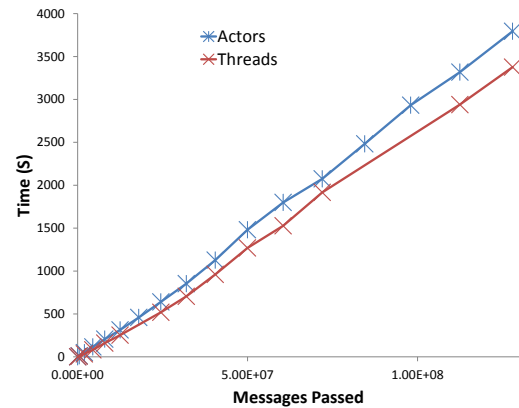


Fig. 8: Execution time scaling as messages passed increases.

by a seemingly random amount. To test this hypothesis, runs were made on a single CPU machine up to 1 million agents (capped due to memory limitations) and no random variation was seen. While this is evidence in support of our hypothesis, the problem merits additional effort to determine the exact causes.

6.3 Computational Workload

The second test was to see how efficiently each framework is for computationally intensive agents. This was done by creating an agent type that computes the first 1000*scaler squares. Our hypothesis was that as agent processor use increases, differences in framework efficiency have less affect on the resulting execution time. This is likely due to the fact that regardless of which framework this agent code is in, it still requires the same amount of CPU time. As Fig. 7 shows, this hypothesis appears to be correct.

6.4 Messaging

Our final test was of message passing performance. This was done using the previously described communicating

agents simulation. In it, each agent is friends with every other agent, so messages passed scales quadratically. This was done instead of numerous rounds of messaging for a small constant number of agents because it seems like a more realistic way in which a framework would be stressed. It is also the case that the threaded framework handles many messages in one round very differently from many rounds of a few messages per step.

As seen in Fig. 8, both frameworks exhibit roughly similar linear scaling with messages sent, but the actor framework scales slightly less optimally. This difference can be explained by the fact that starting and ending steps requires message sending in the actor framework.

6.5 Overall Scaling

The overall performance of actors for MAS simulation in Scala was slightly inferior to that of threads. In terms of memory usage, both frameworks were similar, but the threaded framework had measurably better execution time for the messaging benchmark. It also did not suffer any unusual issues on the agent count benchmark. Since the

primary way a MAS will be scaled up is by increasing the number of agents and message volume, from a purely performance perspective, threads are the obvious way to go.

7. Future Work

There are a number of ways in which our actor framework could be improved. One of the main motivations for using Scala was its extensibility, so a DSL for writing agent AI could be implemented. This would make writing agent code much simpler. Another task would be to use Scala's remote actor library to distribute this framework over multiple machines. Remote actors allow local proxy actors to exist for actors on other machines. These proxies can be sent messages that are forwarded to the actual actor. Other approaches to synchronization could also be employed. The current framework requires that all actors compute in lockstep, but this restriction could be loosened as long as agents were required to maintain the ability to respond to requests for information about their past states. A final task could also be to write a stripped down version of the actor library and tune it for the purpose of MAS simulation. This has the potential to eliminate the performance gap between the thread and actor approaches.

8. Conclusion

We set out to compare the relative performance of two approaches to parallelism when applied to the simulation of multi-agent systems in the language Scala. Our conclusion is that the threaded approach is superior from a strictly performance-oriented point of view. However, with the exception of issues relating to the agent number benchmark, actors did perform respectably. Given the ease of writing the actor framework and the numerous ways it could be improved, such as distribution across multiple machines, loosening of time synchronization, or tuning the actor implementation, this model shows clear promise applied to MAS simulation. The language Scala is also an excellent choice due to the ease with which it can be extended to facilitate a DSL for writing agent AI. For these reasons, we recommend that future groups attempting to implement MAS frameworks consider using Scala actors.

References

- [1] K. P. Sycara, "Multiagent systems," *AI Magazine*, vol. 19, pp. 79–92, 1998.
- [2] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.
- [3] J. Armstrong, "Erlang," *Commun. ACM*, vol. 53, pp. 68–75, September 2010. [Online]. Available: <http://doi.acm.org/10.1145/1810891.1810910>
- [4] "The computer language benchmark game," <http://shootout.alioth.debian.org/>, 3 2011.
- [5] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun, "Language virtualization for heterogeneous parallel computing," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 835–847. [Online]. Available: <http://doi.acm.org/10.1145/1869459.1869527>
- [6] T. Rompf and M. Odersky, "Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls," in *Proceedings of the ninth international conference on Generative programming and component engineering*, ser. GPCE '10. New York, NY, USA: ACM, 2010, pp. 127–136. [Online]. Available: <http://doi.acm.org/10.1145/1868294.1868314>
- [7] C. Hofer and K. Ostermann, "Modular domain-specific language components in scala," *SIGPLAN Not.*, vol. 46, pp. 83–92, October 2010. [Online]. Available: <http://doi.acm.org/10.1145/1942788.1868307>
- [8] M. Odersky, "The scala experiment: can we provide better language support for component systems?" in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 166–167. [Online]. Available: <http://doi.acm.org/10.1145/1111037.1111052>
- [9] A. Law, *Simulation Modeling and Analysis with Expertfit Software*. McGraw-Hill Science/Engineering/Math, 2006.
- [10] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the jvm platform: a comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '09. New York, NY, USA: ACM, 2009, pp. 11–20. [Online]. Available: <http://doi.acm.org/10.1145/1596655.1596658>
- [11] C. Varela, C. Abalde, L. Castro, and J. Gulías, "On modelling agent systems with erlang," in *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, ser. ERLANG '04. New York, NY, USA: ACM, 2004, pp. 65–70. [Online]. Available: <http://doi.acm.org/10.1145/1022471.1022481>
- [12] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.*, vol. 410, pp. 202–220, February 2009. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1496391.1496422>
- [13] —, "Event-Based Programming without Inversion of Control," in *Modular Programming Languages*, ser. Lecture Notes in Computer Science, D. E. Lightfoot and C. A. Szyperski, Eds., 2006, pp. 4–22.
- [14] T. Rompf, I. Maier, and M. Odersky, "Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform," in *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '09. New York, NY, USA: ACM, 2009, pp. 317–328. [Online]. Available: <http://doi.acm.org/10.1145/1596550.1596596>

Asynchronous Communication for Finite-Difference Simulations on GPU Clusters using CUDA and MPI

D. P. Playne¹, and K. A. Hawick¹

¹Computer Science, IIMS, Massey University, Auckland, New Zealand

Abstract—*Graphical processing Units (GPUs) are finding widespread use as accelerators in computer clusters. It is not yet trivial to program applications that use multiple GPU-enabled cluster nodes efficiently. A key aspect of this is managing effective communication between GPU memory on separate devices on separate nodes. We develop a algorithmic framework for Finite-Difference numerical simulations that would normally require highly synchronous data-parallelism so they can effectively use loosely coupled GPU-enabled cluster nodes. We employ asynchronous communications and appropriate memory overlay of computations and communications to hide latency.*

Keywords: GPU; asynchronous communications; clusters; CUDA; MPI

1. Introduction

Accelerators such as Graphical Processing Units (GPUs) have steadily been finding a role in supercomputer systems in recent years and at the time of writing are particularly prominent in major international systems in the Top500 list of Supercomputers [1]. The Tianhe-1A(top), Nebulae(second) and Tsubame(fourth) all employ GPU accelerators and seventeen major systems out of the Top 500 in November 2010 use NVIDIA GPUs to accelerate nodes.

There is importance in understanding how GPU accelerators behave when combined in a multi-processor system for various applications. The Linpack benchmark[2] used by the compilers of the Top 500 list tests capabilities in dense linear algebra. This is certainly an important application paradigm but there are others and we are interested in simulation models on rectilinear in hyper-dimensional systems. A good test application for this paradigm is that of solving finite-difference field equations in one, two, three and higher hyper-dimensional meshes.

In previous work[3] we considered dual GPU acceleration of a single compute node and we have since been able to experiment with triple and quadruple[4] GPUs-per-node as well as a range of different GPU models with varying numbers of cores and memory configurations. In this present paper we study the tradeoff space that arises from decomposing a hyper-dimensional rectilinear

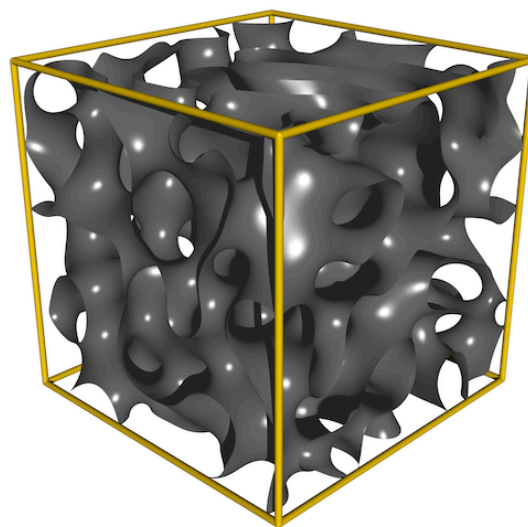


Fig. 1: Ray-traced rendering of a Cahn-Hilliard system simulated on a GPU cluster.

problems such as solving a high-order partial differential equation (PDE) using finite-difference methods [5] on a GPU-accelerated cluster.

Finite-difference methods are still used extensively in computational simulations – particularly in wave-based seismic exploration applications[6] and generally are straightforward to parallelise using geometric stencil methods of decomposition which attain good computational speedup[3], [7] and especially on GPUs [8], [9], [10], [11].

There are various ways to “slice and dice” the data set – and in fact for a typical simulation problem of a physical system one can often choose the system size to best fit the memory configuration and layout of the parallel system. In this paper we report on results for a periodic mesh simulation of the Cahn-Hilliard equation for materials science using a second-order space/second-order-time finite-differencing approach.

There are a number of useful parallel programming technologies that could be employed for these simulations. In this paper we focus on the combination of the open standard Message Passing Interface (MPI)[12], [13] to program inter-node communications and NVIDIA’s Com-

pute Unified Device Architecture (CUDA) [14], [15], [16] programming language for programming the calculations on the GPUs themselves.

A great deal of work has been done on data-parallelism as it pertains to such halo problems[17]. However managing the computation to communications ratio of regular mesh problems on parallel platforms remains a challenge particularly in the case of hybrid architectures. The trade-off space shifts around with the coming of each new parallel platform and we present a study of parameters and (well-known) issues as they pertain to a hybrid system where a multi-dimensional simulation model can be geometrically decomposed across cluster nodes in one of the dimensions, and across the memory and data-parallel thread structure of the accelerating GPU in the other two dimensions.

The use of asynchronous communications across multiple GPU systems is still not yet a widely-known approach and we build on our prior work[3] with multiple GPUs hosted from a single GPU and show the importance of asynchronous communications for a GPU cluster with multiple CPUs accelerated with GPUs. We also include some reference performance data from our example simulation model on combinations of single and multiple GPUs by way of comparison.

The application problem we use as a benchmark for our study is a second-order-space/second-order-time partial differential equation – the Cahn-Hilliard field equation [18], [19], [20], which is used for simulations of phase separation in materials science. Figure 1 shows an example of a simulated Cahn-Hilliard model system, ray-traced to show the spinodally decomposing interfaces between two phases after a long period of computational thermal quenching.

In Section 2 we discuss challenges faced when working with multiple GPU devices. In Section 4 we present methods of decomposing a finite-differencing simulation on multiple GPUs on both a single host and distributed nodes. We present and discuss performance data in Section 5 and offer conclusions in Section 6.

2. Multi-GPU Systems

In previous work [3], [4] we have discussed how the use of CUDA asynchronous memory copies can be used to improve performance when decomposing a finite-difference application across multiple GPUs – connected via PCIe Express bus[21] to a single host CPU. In this present work however we investigate how this method performs when the GPUs are distributed across a compute cluster – when they are hosted by different CPUs.

When GPUs are mounted on a single host, communication between them is relatively simple and very fast. The necessary communication data must be copied from

one device into the host memory and then copied to the appropriate device. When the GPUs are distributed throughout a cluster, this data must be communicated via a network. These networks have significantly higher latency and lower bandwidth, thus we expect a drop in performance. The main advantage of GPUs distributed over a cluster is improved scalability which would otherwise be strictly limited in the case of multiple GPUs contending with one another on a single PCIe bus.

In practice, GPUs hosted on a single machine are currently limited to hosting up to a maximum of four GPUs due to motherboard physical constraints - and sometime also by power-supply and cooling limitations. GPU clusters in contrast have only those physical limitations that arise from network infrastructure. Another advantage of increased GPU numbers is the resulting increased upper bound on feasible model system size.

GPUs still have very limited device memory which restricts the maximum system size that can be simulated. By decomposing the simulation across many GPUs, this maximum simulated system size can be increased considerably. It is an important result for GPU clusters to explore the scalability and locate the higher limitations for a cluster system.

GPUs come in a number of models and variations. In this article, we have focused exclusively on NVIDIA GPUs running the Compute Unified Device Architecture (CUDA) software. Other software systems such as the open compute language standard OpenCL are feasible and promise functionality on other vendor's platforms. Our experience has been that CUDA still delivers considerably more performance than OpenCL and we focus solely on CUDA and NVIDIA devices in this paper. CUDA is not totally trivial to port applications source code to, but we employ a well-optimised and tested source code we developed for solving the Cahn-Hilliard equation.

3. Cahn-Hilliard Equation

We and other authors have described this equation in detail elsewhere[18], [19], [20], but for completeness we give a brief summary here. The equation is usually formulated as:

$$\frac{\partial u}{\partial t} = M \nabla^2 (-Bu + Uu^3 - K \nabla^2 u) \quad (1)$$

The field $u(x, y)$ or $u(x, y, z)$ is a multi-dimensional scalar field taking values between ± 1 which represent the two extreme materials phases. So these might represent different atomic species in an alloy or a two separate sorts of fluids. The field is initialised randomly then “numerically quenched” by stepping it forwards in time. The parameters: M, U, K specify the detailed material properties and for our benchmarking purposes here can

be set to unity. The model then has a single remaining parameter B which controls the temperature of the simulated quench experiment and thus the rate at which the field separates out into domains. The spatial calculus in the equation employs a double Laplacian operator and thus has a larger halo boundary – or number of neighbours – than simpler common finite-difference equations featuring in mathematics textbooks. The Laplacians are approximated by finite-difference stencils to second-order accuracy and the time-stepping must generally be second-order for domain-growth without introducing artifacts from numerical instability[22]. Figure 1 shows a three dimensional model system that has been time-stepped for many iterations after its random quench and exhibits a complex pattern of interleaving spatial component clusters. The figure was rendered as hyper-surfaces that represent the interfaces between physical domains and are shown rendered using ray-tracing.

4. Implementation

To split a finite-differencing simulation of a field equation like the Cahn-Hilliard system across a cluster of GPUs, the field must be decomposed into sections that can be stored and processed by each GPU. There are many options in terms of field decomposition - blocks, layers and so forth[23]. We have found that decomposing the field into layers in the highest dimension has proved the most efficient. The field is split into equal layers and spread across the GPUs in each node of the GPU cluster. This method of decomposition is shown in Figure 2.

The main challenge of decomposing an application across a compute cluster is the relatively high latency of communication across the network. This communication is especially important as the GPU accelerated nodes have a higher computational throughput. In our previous work [3], [4] on multiple GPUs on a single host, our update algorithm communicated the bordering information using asynchronous memory copies to hide latency. This algorithm can be seen in Algorithm 1.

Algorithm 1 Multi-GPU update algorithm using asynchronous memory communication to hide latency.

```

for all steps do
  Compute border cells (stream 1)
  Compute remaining cells in field (stream 2)
  Copy borders from GPU to host (stream 1)
  Exchange borders with neighbours (CPU)
  Copy new borders into GPU (stream 1)
  Synchronize streams
end for

```

The main advantage of this algorithm for multiple GPUs is that the communication can be performed while the

GPU is still working. The main computation time is taken up computing the simulation for the main body of the field. By computing the borders first and using asynchronous memory copies, the communication can be performed during this main computation time.

4.1 Algorithm A

This algorithm can be adapted for use with distributed GPUs. Instead of exchanging borders with another thread through the host memory, this data must be sent via MPI. This algorithm still allows information to be communicated while the GPU is still computing, however the latency of sending data across a network is much higher than simple memory copies and we expect a drop in performance. The adapted algorithm is shown in Algorithm 2.

Algorithm 2 GPU cluster update algorithm using asynchronous memory communication and MPI.

```

for all steps do
  Compute border cells (stream 1)
  Compute remaining cells in field (stream 2)
  Copy borders from GPU to host (stream 1)
  Send data to each neighbour (MPI)
  Wait for data from neighbours (MPI)
  Copy new borders into GPU (stream 1)
  Synchronize streams
end for

```

While this algorithm does work, we found that the performance was unpredictable when the number of hosts was increased. This performance data is presented in Section 5. However, a simple modification to this algorithm has provided faster and more reliable performance.

4.2 Algorithm B

This method uses a uni-directional communication method to exchange borders between GPU nodes. Rather than sending an receiving one border from each neighbour, this method sends data to only one neighbour and only receives data from the other. The amount of data sent is remains the same (each send is twice the size of Algorithm A) but the communication is simpler as the nodes are not trying to send and receive data from the same neighbour. The two methods of communication can be seen in Figure 3. The steps of this algorithm can be seen in Algorithm 3.

This method of uni-directional communication effectively means that the field section each node is responsible is continuously moving through the field. By shifting the data each time-step to accommodate the incoming data, the field is still correct and valid. This method provides more reliable performance and better scalability as the number of cluster nodes increases. These results are presented in Section 5.

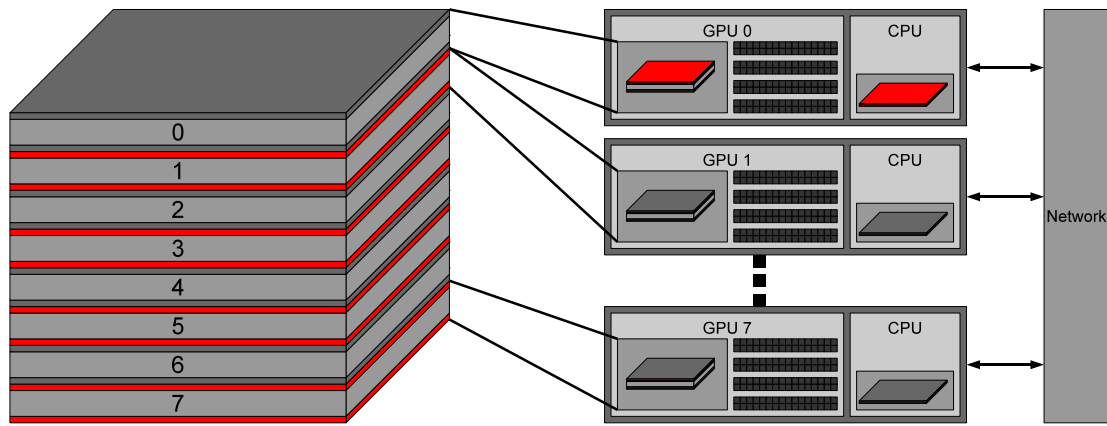


Fig. 2: Layer decomposition of a three-dimensional field split across eight nodes in a GPU cluster.

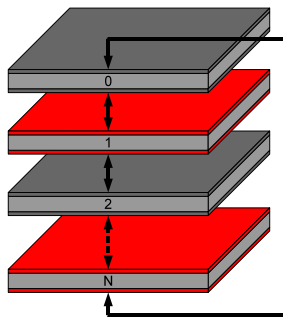


Fig. 3: The Bi-directional border communication scheme as used by Algorithm A.

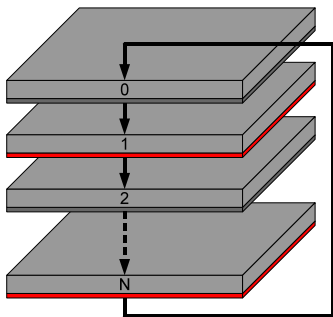


Fig. 4: The Uni-directional border communication scheme as used by Algorithm B.

5. Results and Discussion

To compare these different architectures and algorithms we have measured their performance across a range of system sizes and configurations and compared these to single-GPU and single-host multi-GPU implementations. First we investigate the performance on a GPU cluster of the Bi-Directional (algorithm A) implementation. This performance data can be seen in Figure 5.

As can be seen from the plot, this algorithm shows

Algorithm 3 GPU cluster update algorithm using asynchronous memory communication and MPI.

for all steps do

 Compute border cells (stream 1)
 Compute remaining cells in field (stream 2)
 Copy border from GPU to host (stream 1)
 Send data to one neighbour (MPI)
 Wait for data other neighbour (MPI)
 Copy new border into GPU (stream 1)
 Synchronize streams

end for

varying results and unreliable performance. The algorithm also does not scale well and some systems actually take longer with more nodes. However, this algorithm does allow a larger number of nodes to be utilised (our cluster is limited to 16) and thus allows larger systems to be simulated. Finally we compare our uni-directional communication algorithm (algorithm B) with single GPU data (See Figure 6).

It can be clearly seen that this algorithm provides much more reliable and scalable performance. While it does not quite reach almost linear speed-up attained by the single-host algorithm, it does give a consistent performance improvement and can be scaled to many more GPUs. Considering the increased latency due to network communication, this algorithm provides an efficient method of decomposing a finite-differencing equation across a GPU cluster.

Another point of importance is the maximum system size each architecture is able to support. Finite-differencing systems have a somewhat limited maximum system size due to the relatively small amount of device memory available on GPUs. The most GPUs that the field can be split between, the more device memory available and the larger the maximum system size is. Table 1 shows

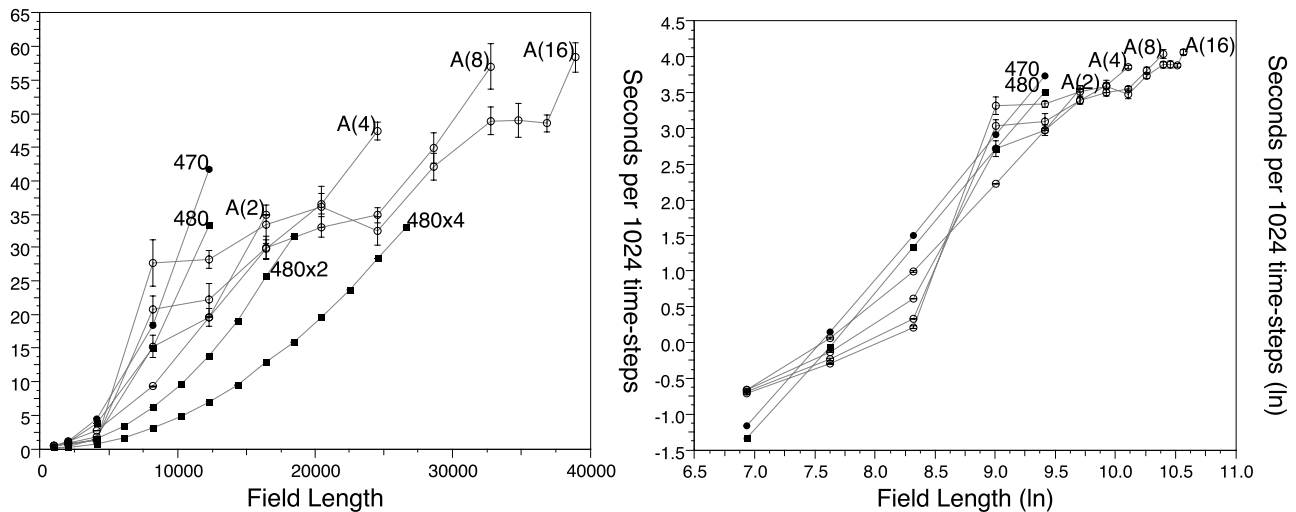


Fig. 5: Comparison of single GPU data and 2, 4, 8 and 16 GTX470s on a distributed cluster using the bi-directional update algorithm (A). Results shown on a linear scale (left) and ln-ln scale (right).

the maximum system size each test architecture is able to compute.

Table 1: Maximum system size that can be simulated on each machine configuration.

Architecture	Max System Size
GTX470	12288^2
GTX480	12288^2
GTX470x2	16384^2
GTX480x2	18432^2
GTX470x4	24576^2
GTX480x4	26624^2
GTX470x8	32768^2
GTX470x16	47104^2

We have also investigated how these update algorithms perform for three-dimensional systems. The single-host algorithm provides the almost linear speedup as seen previously in the two-dimensional version. However, the GPU cluster algorithms are somewhat disappointing. The bi-directional communication algorithm showed the same unreliable results as experienced in two-dimensions but did provide a small speedup over a single GPU.

The uni-directional algorithm close to a 2x speedup over a single GPU with two nodes, however this performance gain does not scale. Simulations decomposed over 4, 8 and 16 nodes showing no further performance gain (and in some cases a slight performance loss). In three-dimensions the communication time outweighs the computational gain of more processing nodes.

The only real advantage decomposing a simulation over more cluster nodes is the maximum system size that can be computed. The single GPU (GTX480) implementation was

limited to 448^3 , four GPUs on a single host was capable of simulating a system of size 896^3 while 16 cluster nodes were capable of computing up to a system size of 1280^3 .

6. Summary and Conclusions

We have reported on geometric domain decomposition results of a finite difference application problem (the Cahn-Hilliard partial differential equation) on a cluster of NVIDIA GPU-accelerated Intel CPUs. We have explored combinations of multiple GPUs on a single node as well as a cluster of single-GPU nodes. We have presented and compared two algorithms for managing communication between these nodes and compared their performance.

We have shown that the increased complexity involved when communicating across a network interconnect can cause unexpected and unreliable results as exhibited by the bi-directional communication update algorithm. Even minor changes such as the adoption of a uni-directional communication method can have drastic impact on the reliability and overall performance of the simulation.

The algorithm we have presented makes efficient use of distributed GPU nodes showing good scalability and improvements in both maximum system size and performance for two-dimensional simulations. It also shows a limited but tangible performance gain in three-dimensions but more importantly allows larger simulated systems to be computed than would be otherwise feasible in the memory of a single CPU/GPU combination.

GP-GPU computing has already offered a new lease of life to data-parallel computing as an accelerator for individual CPUs. We anticipate it will now continue to offer good means of accelerating cluster systems at the small to medium commodity priced range using the “gamer”

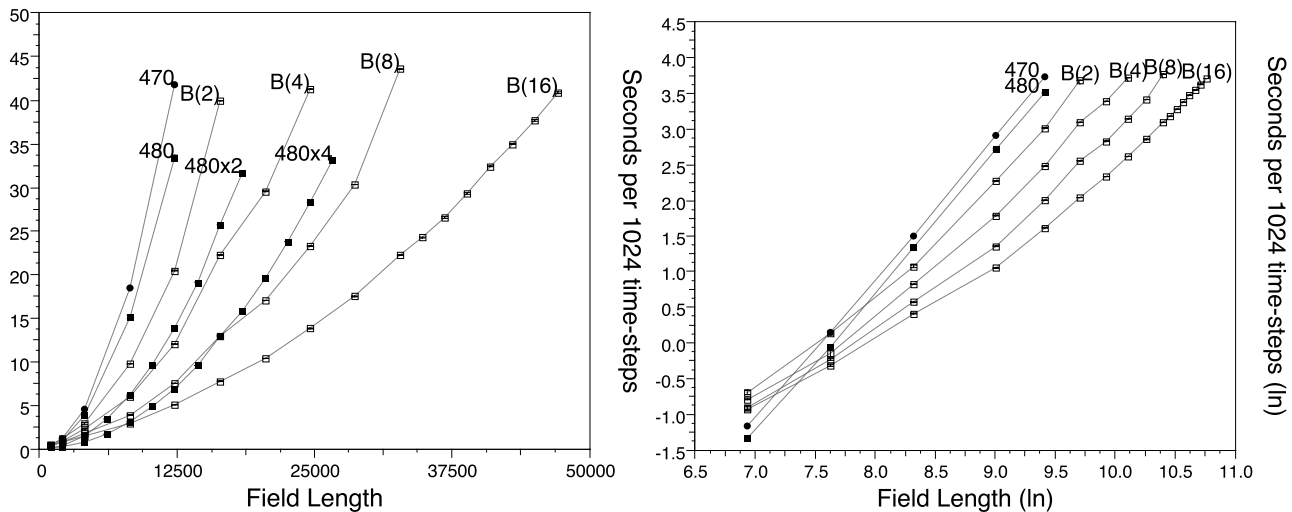


Fig. 6: Comparison of single GPU data and 2, 4, 8 and 16 GTX470s on a distributed cluster using the uni-directional update algorithm (B). Results shown on a linear scale (left) and ln-ln scale (right).

grade GPUs we have discussed here as well as blade grade systems used in supercomputers.

Acknowledgments

Thanks to A.Leist for valuable assistance with configuring MPI.

References

- [1] H. Meuer, E. Strohmaier, H. Simon, and J. Dongarra, "36th list of top 500 supercomputer sites," www.top500.org/lists/2010/11/press-release, November 2010.
- [2] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK Benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003.
- [3] D. Playne and K. Hawick, "Hierarchical and Multi-level Schemes for Finite Difference Methods on GPUs," in *Proc. CCGrid 2010, Melbourne, Australia*, no. CSTN-099, May 2010.
- [4] D. P. Playne and K. A. Hawick, "Comparison of GPU Architectures for Asynchronous Communication with Finite-Differencing Applications," Massey University, Tech. Rep. CSTN-111, 2010, submitted to: *Concurrency and Computation: Practice and Experience*.
- [5] A. Mitchell and D. Griffiths, *The Finite Difference Method in Partial Differential Equations*. Wiley, 1980, no. ISBN 0-471-27641-3.
- [6] B. Alessandrini and V. Raganelli, "The propagation of acoustic and elastic waves in a heterogeneous discrete medium," *Eur.J.Mech., A/ Solids*, vol. 11, no. 4, pp. 519–538, 1992.
- [7] R. F. Barrett, P. C. Roth, and S. W. Poole, "Finite difference stencils implemented using chapel," Oak Ridge National Laboratory, Tech. Rep. ORNL Technical Report TM-2007/122, 2007.
- [8] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, no. ISBN:978-1-60558-517-8, 2009.
- [9] D. Egloff, "High Performance Finite Difference PDE Solvers on GPUs," QuantAlea GmbH, Tech. Rep., 2010.
- [10] S. E. Krakowski, L. E. Turner, and M. M. Okoniewski, "Acceleration of Finite-Difference Time-Domain (FDTD) Using Graphics Processor Units (GPU)," *IEEE MIT-S Digest*, vol. WEIF-2, pp. 1033–1036, 2004.
- [11] S. Zainud-Deen, E. El-Deen, M. Ibrahim, K. Awadalla, and A. Botros, "Electromagnetic Scattering Using GPU-Based Finite Difference Frequency Domain Method," *Prog. in Electromagnetics Res. B*, vol. 16, pp. 351–369, 2009.
- [12] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994, ISBN 0-262-57104-8.
- [13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*. Argonne National Laboratories, 1996.
- [14] *CUDA™ 3.1 Programming Guide*, NVIDIA® Corporation, 2010, last accessed August 2010. [Online]. Available: <http://www.nvidia.com/>
- [15] W.-M. Hwu, C. Rodrigues, S. Ryoo, and J. Stratton, "Compute Unified Device Architecture Application Suitability," *Computing in Science and Engineering*, vol. 11, pp. 16–26, 2009.
- [16] A. Leist, D. Playne, and K. Hawick, "Exploiting Graphical Processing Units for Data-Parallel Scientific Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 2400–2437, December 2009, CSTN-065.
- [17] K. S. Perumalla and B. G. Aaby, "Data parallel execution challenges and runtime performance of agent simulations on gpus," in *SpringSim '08: Proceedings of the 2008 Spring simulation multicongference*. New York, NY, USA: ACM, 2008, pp. 116–123.
- [18] J. W. Cahn and J. E. Hilliard, "Free Energy of a Nonuniform System. I. Interfacial Free Energy," *The Journal of Chemical Physics*, vol. 28, no. 2, pp. 258–267, 1958.
- [19] K. A. Hawick and D. P. Playne, "Modelling and visualizing the Cahn-Hilliard-Cook equation," in *Proceedings of 2008 International Conference on Modeling, Simulation and Visualization Methods (MSV'08)*, Las Vegas, Nevada, July 2008.
- [20] D. Playne and K. Hawick, "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.
- [21] PCI-SIG, "PCIe Express Base Specification 1.1," <http://www.pcisig.com/specifications/pciexpress/base>, November 2010.
- [22] K. A. Hawick, "Domain Growth in Alloys," 1991, Edinburgh University, Ph.D. Thesis.
- [23] D. A. Reed, L. M. Adams, and M. L. Patrick, "Stencils and problem partitionings: Their influence on the performance of multiple processor systems," *IEEE Transactions on Computers C*, vol. 36, no. 7, pp. 845–858, jul 1987.

An Efficient Computational Approach for Solving a Class of Nonlinear Integral Equations

K. Maleknejad and P. Torabi

School of Mathematics, Iran University of Science & Technology
Narmak, Tehran 16846 13114, Iran

Abstract—Most of the methods, which used to solve nonlinear integral equations, transform the equation into a system of nonlinear equations. It is cumbersome to solve these systems, or the solution may be unreliable. In this paper, we propose an iterative approach based on fixed point method and Sinc quadrature, which has exponentially rate of convergence, to solve Hammerstein integral equation. Convergence of numerical scheme is proved by some theorems and some numerical examples are given to show applicability and accuracy of numerical method.

Keywords: Hammerstein integral equation; Fixed point method; Sinc quadrature.

1. Introduction

Many problems arise in the mathematical physics, engineering, biology, economics and so on lead to mathematical models described by nonlinear integral equations. (cf. [1], [2], [3]). For instance, the Hammerstein integral equations appear in nonlinear physical phenomena such as electromagnetic fluid dynamics, reformulation of boundary value problems with a nonlinear boundary condition (see [4]). This equation is as following

$$x(t) = g(t) + \int_a^b k(t, \tau)H(\tau, x(\tau))d\tau, \quad (1)$$

for all $t \in I = [a, b]$ which is a Fredholm-type integral equation. Many different methods have been used to approximate a solution for these integral equations. For example we can mention the following approaches. In [5], a variational of Nystrom's method is introduced. The classical method of successive approximations used in [6]. Some collocation-type methods developed in [7], [8]. An approach based on single-term Walsh series proposed in [9]. In [10] Hammerstein equation was solved using Walsh-Hybrid functions. Some methods based on interpolations, Petrov-Galerkin, combination of spline-collocation and Lagrange interpolation, and Dabechies wavelets introduced in [11], [12], [13], [14].

In the methods mentioned above, the integral equation is transformed into a system of nonlinear algebraic equations which has to be solved with iterative methods. It is cumbersome to solve these systems, or the solution may be unreliable. To eliminate this problem, we try to prepare

a numerical scheme to approximate a solution for integral equation (1) based on fixed point method and some quadrature rules such as Sinc quadrature where it has exponential rate of accuracy [15], [16]. This method has two advantages that encourages us to use it. In one hand, there is not any system of nonlinear equations with its difficulties. On the other hand, this method is very simple to apply and to make an algorithm.

The organization of this paper is as follows. First we mention some necessary concepts such as contractive operators and Sinc quadrature which we will use later. Then we introduce our numerical technique, and discuss its convergence. Finally, we present some numerical examples to show efficiency and accuracy of numerical method.

2. Preliminaries

Let us introduce some necessary concepts and tools which help us to frame our method. They can be found in numerical analysis books such as [15]-[18].

2.1 Contractive operator in Banach spaces

Let V be a Banach space with the norm $\|\cdot\|_V$ and let K be a subset of V . Consider an operator $T : K \rightarrow V$ defined on K .

Definition 2.1 We say an operator $T : K \rightarrow V$ is contractive with contractivity constant $\alpha \in [0, 1)$ if

$$\|T(u) - T(v)\|_V \leq \alpha \|u - v\|_V, \quad \forall u, v \in K.$$

The operator T is called non-expansive if

$$\|T(u) - T(v)\|_V \leq \|u - v\|_V, \quad \forall u, v \in K,$$

and Lipschitz continuous if there exists a constant $L \geq 0$ such that

$$\|T(u) - T(v)\|_V \leq L \|u - v\|_V, \quad \forall u, v \in K.$$

The following theorem is known as Banach fixed point theorem and play an important role to guarantee existence and uniqueness of the solution of nonlinear equations.

Theorem 2.2 Assume that K is a nonempty closed set in a Banach space V , and further, that $T : K \rightarrow K$ is a

contractive mapping with contractivity constant $0 \leq \alpha < 1$. Then the following results hold

- (1) Existence and uniqueness: There exists a unique $u \in K$ such that

$$u = T(u).$$

- (2) Convergence and error estimates of the iteration: For any $u_0 \in K$, the sequence $\{u_n\} \subset K$ defined by $u_{n+1} = T(u_n), n = 0, 1, \dots$, converges to u :

$$\|u_n - u\|_V \rightarrow 0, \text{ as } n \rightarrow \infty.$$

For the error, the following bounds are valid

$$\begin{aligned} \|u_n - u\|_V &\leq \frac{\alpha^n}{1 - \alpha} \|u_0 - u_1\|_V, \\ \|u_n - u\|_V &\leq \frac{\alpha}{1 - \alpha} \|u_{n-1} - u_n\|_V, \\ \|u_n - u\|_V &\leq \alpha \|u_{n-1} - u\|_V. \end{aligned} \tag{2}$$

Proof: [17]

This theorem is called by a variety of names in the literature, with the contractive mapping theorem another popular choice. It is also called Picard iteration in settings related to differential equations.

2.2 Sinc quadrature

We introduce the cardinal function and some of its quadrature properties. For this result $sinc(x)$ definition is followed by

$$sinc(x) = \begin{cases} \frac{\sin(\pi x)}{\pi x}, & x \neq 0 \\ 1, & x = 0, \end{cases}$$

where has the following graph

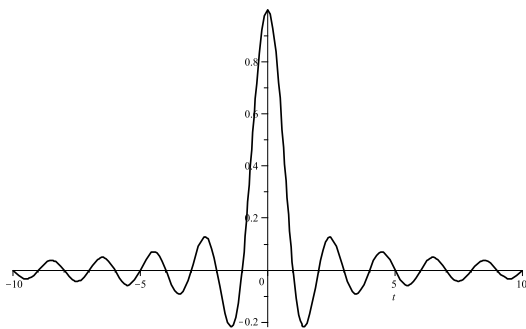


Fig. 1: Graph of Function $sinc(x)$

Now, for $h > 0$ and integer k , we define k 'th Sinc function with step size h by

$$S(k, h)(x) = \frac{\sin(\pi(x - kh)/h)}{\pi(x - kh)/h}.$$

Let $D_d = \{z \in \mathcal{C} : |Im(z)| < d\}$ and $t = \phi(z)$ denote a conformal map which maps the simply connected domain D with boundary ∂D onto a strip region D_d such that

$$\phi((a, b)) = (-\infty, \infty),$$

$$\lim_{t \rightarrow a} \phi(t) = -\infty,$$

$$\lim_{t \rightarrow b} \phi(t) = \infty.$$

Now, in order to have the Sinc approximation on a finite interval (a, b) conformal map is employed as $\phi(x) = \ln(\frac{x-a}{b-x})$. This map carries the eye-shaped complex domain

$$\left\{ z = x + iy : \left| \arg\left(\frac{z-a}{b-z}\right) \right| < d \leq \frac{\pi}{2} \right\},$$

onto the infinite strip $D_d = \{\mu = \alpha + \beta i : |\beta| < d < \frac{\pi}{2}\}$, and basis function on finite interval (a, b) are given by

$$S(k, h) \circ \phi(x) = \frac{\sin(\pi(\phi(x) - kh)/h)}{\pi(\phi(x) - kh)/h}.$$

Now, we use the following sinc quadrature [16] formulas to discrete an integral by

$$\int_a^b f(z) dz = h \sum_{k=-N}^N \frac{f(z_k)}{\phi'(z_k)} + O(\exp(-\frac{2\pi d N}{\log(\frac{2\pi d N}{\beta})})) \tag{3}$$

where $z_k = \frac{a+b}{1+e^{kh}}$ for $k = -N, \dots, N$ and $h = \frac{1}{N} \log(\frac{\pi d N}{\beta})$. We can see in equation (3) which this quadrature rule has exponentially rate of convergence.

3. The proposed approach

We consider Hammerstein integral equation (1)

$$x(t) = g(t) + \int_a^b k(t, \tau) H(\tau, x(\tau)) d\tau,$$

and assume that $g \in C[a, b]$ and $k \in L^2[a, b]^2$. Now we define operator T as following

$$(Tx)(t) = g(t) + \int_a^b k(t, \tau) H(\tau, x(\tau)) d\tau. \tag{4}$$

Obviously, the solution of equation (1) is the fixed point of operator T . By choosing the initial function $x_0(t) \in C[a, b]$ we can introduce the fixed point iteration

$$\begin{aligned} x_{n+1}(t) &= (Tx_n)(t) \\ &= g(t) + \int_a^b k(t, \tau) H(\tau, x_n(\tau)) d\tau, \end{aligned} \tag{5}$$

for all $t \in [a, b]$ and $n \geq 0$.

In the following theorem, we show that under proper assumptions, T is a contractive operator and then it has a unique fixed point. Consequently, the sequence $\{x_n(t)\}_{n=0}^\infty$ generated by iteration (5) converges to this unique fixed point of T .

Theorem 3.1 Consider the operator T introduced by relation (4) and assume $g \in C[a, b]$, $k \in L^2[a, b]^2$ i.e. there exists a constant $M > 0$ where

$$\left(\int_a^b k^2(t, \tau) d\tau \right)^{\frac{1}{2}} \leq M < \infty.$$

Also $H(\tau, x)$ satisfies a uniform Lipschitz condition with respect to its second argument

$$\|H(\tau, x) - H(\tau, y)\| \leq h\|x - y\|, \quad (6)$$

For all $\tau \in [a, b]$ and $x, y \in \mathbb{R}$.

Moreover, assume

$$Mh(b - a)^{\frac{1}{2}} < 1. \quad (7)$$

Then T transforms $C[a, b]$ into itself and it is a contractive operator on $C[a, b]$.

Proof: Suppose $x, y \in C[a, b]$, then for all $t \in [a, b]$ we derive the following inequality

$$\begin{aligned} & |(Tx)(t) - (Ty)(t)| \\ &= \left| \int_a^b k(t, \tau) (H(\tau, x(\tau)) - H(\tau, y(\tau))) d\tau \right| \\ &\leq \left(\int_a^b k^2(t, \tau) d\tau \right)^{\frac{1}{2}} \|H(\tau, x) - H(\tau, y)\| (b - a)^{\frac{1}{2}} \\ &\leq Mh(b - a)^{\frac{1}{2}} \|x - y\|, \end{aligned} \quad (8)$$

where Cauchy-Schwartz inequality and Lipschitz condition (6) used in recent relation. From the above estimate we drive the following inequality

$$\|Tx - Ty\| \leq Mh(b - a)^{\frac{1}{2}} \|x - y\|$$

which yields the continuity of the operator T , so it transforms $C[a, b]$ into itself

$$T : C[a, b] \rightarrow C[a, b].$$

Moreover, by assumption (7), recent inequality yields operator T is contractive with contractivity constant $\alpha = Mh(b - a)^{\frac{1}{2}} < 1$. \square

According to the Banach fixed point theorem (2.2), the operator T has a unique fixed point in $C[a, b]$, and by iteration method (5) the generated sequence $\{x_n\}_{n=0}^{\infty}$ is converges to the fixed point x . The error bound is introduced in theorem (2.2) i.e.

$$\|x_n - x\| \leq \frac{\alpha^n}{1 - \alpha} \|x_0 - x_1\|. \quad (9)$$

Now, to start the iteration (5) we need an initial function. Since $g(t) \in C[a, b]$ we can choose it as the initial function $x_0 \equiv g$. In each iteration we have to calculate the integral part of operator T . It can be cumbersome if we compute that integral analytically, so we use a quadrature method such

as Sinc integration to evaluate integral part of operator T numerically. By substituting Sinc quadrature (3) in equation (5) we have

$$\begin{aligned} x_{n+1}(t) &= (Tx_n)(t) \\ &= g(t) + \int_a^b k(t, \tau) H(\tau, x_n(\tau)) d\tau \\ &\approx g(t) + h \sum_{k=-N}^N \frac{k(t, z_k) H(z_k, x_n(z_k))}{\phi'(z_k)}, \end{aligned} \quad (10)$$

where z_k , h and $\phi(t)$ introduced in subsection (2.2).

Let $x_{n+1}^{(N)}(t)$ be the approximation of $x_{n+1}(t)$ by Sinc quadrature in (10) and $x(t)$ be the exact fixed point of T . Then we have the following error bound

$$\|x_{n+1}^{(N)} - x\| \leq \|x_{n+1}^{(N)} - x_{n+1}\| + \|x_{n+1} - x\|,$$

where the bound of these errors obtained in (3) and (9).

In iterative equation (10) in each iteration, x_{n+1} arises directly from x_n without solving any large system of nonlinear algebraic equations. It is a great advantage of the proposed method. In the next section accuracy and efficiency of the method have been showed by some numerical examples.

4. Experimental results

In order to test the utility of the proposed numerical method, we give the following examples. In all examples we choose the tolerance $\varepsilon = 10^{-7}$ to stop the iterations, i.e. fixed point iterations stop when $\|x_n - x_{n-1}\| < \varepsilon$. All routines have been written in Fortran 90.

Example 4.1 Consider the following Hammerstein integral equation

$$x(t) = e^t - t \sin(t) + \int_0^1 e^{-2\tau} \sin(t) x^2(\tau) d\tau, \quad 0 \leq t \leq 1.$$

The exact solution of this equation is $x_{exact} = e^t$. We solve it by our proposed method. Table (1) shows maximum error and number of iterations for some different N . Exact and approximation solutions based on Sinc quadrature rule with $N = 20$ and $N = 50$ is shown in Figure (2).

Table 1: Numerical Results for Example 4.1

N	$\ e\ _{\infty}$	IT
20	7.978E - 7	25
50	4.129E - 7	25
100	1.858E - 7	25

Example 4.2 Consider the following boundary value problem

$$\begin{aligned} x''(t) - e^{x(t)} &= 0, \quad 0 \leq t \leq 1, \\ x(0) = x(1) &= 0, \end{aligned} \quad (11)$$

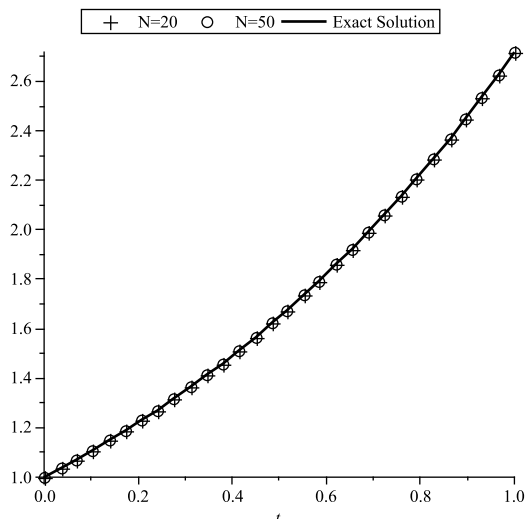


Fig. 2: Approximation and Exact Solutions using Sinc Quadrature with $N = 20$ and $N = 50$ for Hammerstein Integral Equation Solved in Example 4.1

which is of great interest in hydrodynamics [19]. This problem has the unique solution

$$x_{exact}(t) = -\ln(2) + \ln(\lambda(t))$$

where

$$\lambda(t) = \left(\frac{c}{\cos(\frac{1}{2}c(t - \frac{1}{2}))} \right)^2.$$

Here, c is the root of the equation

$$\left(\frac{c}{\cos(\frac{c}{4})} \right)^2 = 2.$$

problem (11) can be reformulated as the integral equation

$$x(t) = \int_0^1 k(t, \tau) e^{x(\tau)} d\tau, \quad 0 \leq t \leq 1,$$

where

$$k(t, \tau) = \begin{cases} -\tau(1-t), & \tau \leq t, \\ -t(1-\tau), & t \leq \tau, \end{cases}$$

Table (2) shows maximum error and number of iterations for some different N . Exact and approximation solutions based on Sinc quadrature rule with $N = 20$ and $N = 50$ is shown in Figure (3).

Table 2: Numerical Results for Example 4.2

N	$\ e \ _{\infty}$	IT
20	$2.051E - 5$	6
50	$3.370E - 6$	7
100	$9.182E - 7$	7

Example 4.3 In this example we consider the mathematical

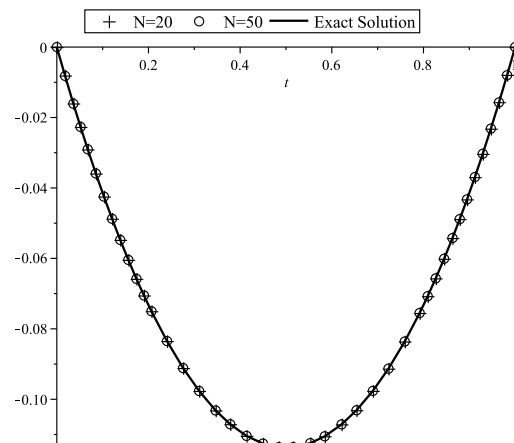


Fig. 3: Approximation and Exact Solutions using Sinc Quadrature with $N = 20$ and $N = 50$ for Hammerstein Integral Equation Solved in Example 4.2

model for an adiabatic tubular chemical reactor discussed (see [20]), which in the case of steady state solutions, can be stated as the ordinary differential equation

$$x''(t) - \lambda x'(t) + F(\lambda, \mu, \beta, x(t)) = 0, \quad 0 \leq t \leq 1,$$

with boundary conditions

$$x'(0) = \lambda x(0), \quad x'(1) = 0,$$

where

$$F(\lambda, \mu, \beta, x(t)) = \lambda \mu (\beta - x(t)) e^{x(t)}.$$

The problem can be converted into a Hammerstein integral equation of the form [20]

$$x(t) = \int_0^1 k(t, \tau) H(\tau, x(\tau)) d\tau, \quad 0 \leq t \leq 1, \quad (12)$$

where $k(t, \tau)$ is defined by

$$k(t, \tau) = \begin{cases} 1, & \tau \leq t, \\ e^{\lambda(t-\tau)}, & t \leq \tau, \end{cases}$$

and

$$H(\tau, x(\tau)) = \mu (\beta - x(\tau)) e^{x(\tau)}.$$

The existence and uniqueness of the solution for this Hammerstein integral equation with respect to the values of parameters λ , μ and β is given in [20]. In [19] a composite collocation method is used to solve the integral equation (12) for the particular values of the parameters $\lambda = 10$, $\mu = 0.02$, and $\beta = 3$ which guarantee the existence and uniqueness of the solution for this integral equation. We solved it by proposed method for $N = 100$ with obtained maximum error $\| e \|_{\infty} = 1.118E - 8$ after 4 iterations. Table (3) shows a comparison between the numerical results in some points of the interval $[0, 1]$ obtained by proposed

Table 3: Numerical Results for Example 4.3

t	composite collocation	proposed method
0.0	0.0060483739	0.0060533930
0.2	0.0181929364	0.0181980300
0.4	0.0304246702	0.0304298400
0.6	0.0426691183	0.0426743000
0.8	0.0543716533	0.0543762800
1.0	0.0614587374	0.0614589000

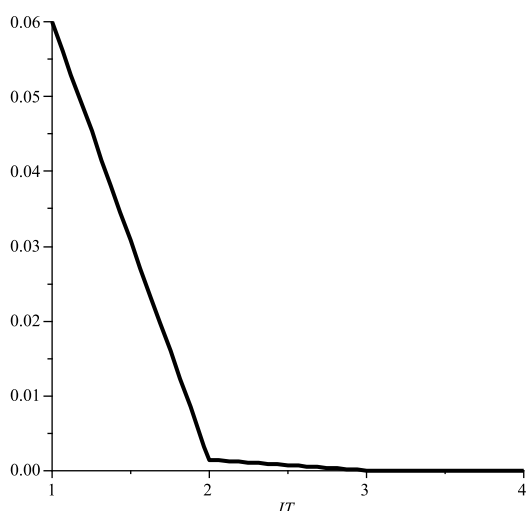


Fig. 4: Maximum Error Related to Each Iteration of Fixed Point Method in Example 4.3

method and the composite collocation method in [19].

Example 4.4 The exact solution of following Hammerstein integral equation is $x_{exact} = \sin(t)$:

$$x(t) = \frac{1}{2} \sin(t) - \frac{1}{4} \cos(t) + \frac{1}{4} \cos(t - 2) + \int_0^1 \cos(t - \tau)x(\tau)d\tau, \quad 0 \leq t \leq 1.$$

By applying the proposed method we approximate the solution of this equation. Table (4) shows maximum error and number of iterations for some different N . Exact and approximation solutions based on Sinc quadrature rule with $N = 20$ and $N = 50$ is shown in Figure (5).

Table 4: Numerical Results for Example 4.4

N	$\ e\ _\infty$	IT
20	2.138E-3	164
50	3.440E-4	166
100	8.640E-5	171

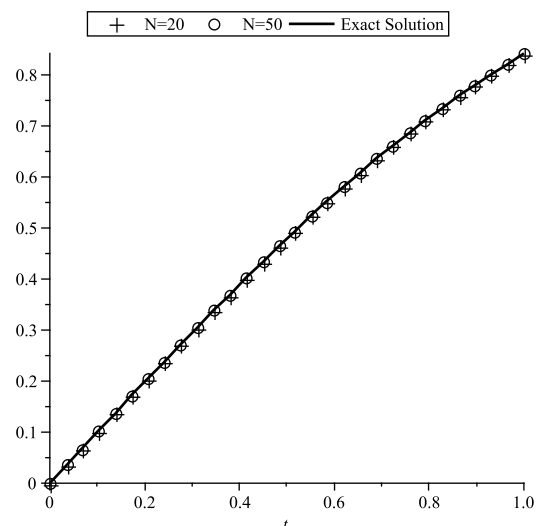


Fig. 5: Approximation and Exact Solutions using Sinc Quadrature with $N = 20$ and $N = 50$ for Hammerstein Integral Equation Solved in Example 4.4

5. Conclusion

We use an iterative method based on fixed point technique and quadrature rule to find the approximate solution of Hammerstein integral equation. Using this method, a sequence of functions is obtained which is proved its convergence to the exact solution. This method has two advantages that encourages to use it. In one hand, there is not any system of nonlinear equations with its difficulties. On the other hand, this method is very simple to apply and to make an algorithm. Numerical results are verified that the method employed in the paper is valid. It is worthy to note that this method can be used as a very accurate algorithm for solving linear and nonlinear integro-differential equations and functional integral equation arising in physics and other fields of applied mathematics.

References

- [1] D. O'Regan and M. Meehan, *Existence theory for nonlinear integral and integro-differential equations*, Kluwer Academic, Dordrecht, 1998.
- [2] R. P. Agarwal, D. O'Regan and P. J. Y. Wong, *Positive Solutions of Differential, Difference and Integral Equations*, Kluwer Academic, Dordrecht, 1999.
- [3] T. A. Burton, *Volterra Integral and Differential Equations*, Academic Press, New York, 1983.
- [4] K. E. Atkinson, *The numerical solution of integral equations of the second kind*, Cambridge University Press, Cambridge, 1997.
- [5] L. J. Lardy, "A variation of Nystrom's method for Hammerstein equations," *J. Integral Equations*, vol. 3, No. 1, pp. 43-60, 1981.
- [6] F. G. Tricomi, *Integral Equations*, Dover Publications, New York, 1985.
- [7] K. Kumar and I. H. Sloan, "A new collocation-type method for Hammerstein integral equations," *Math. Comp.*, vol. 48, No. 178, pp. 585-593, 1987.
- [8] H. Brunner, "Implicitly linear collocation methods for nonlinear Volterra equations," *Appl.Numer. Math.*, vol. 9, No.3-5, pp. 235-247, 1992.

- [9] B. Sepehrian and M. Razzaghi, "Solution of nonlinear Volterra-Hammerstein integral equations via single-term Walsh series method," *Mathematical Problems in engineering*, vol. 5, pp. 547–554, 2005.
- [10] Y. Ordokhani, "Solution of Fredholm-Hammerstein integral equations with Walsh-Hybrid functions," *International Mathematical Forum*, vol. 4, No.20, pp. 969–976, 2009.
- [11] K. Maleknejad, M. Karami and N. Aghazadeh, "Numerical Solution of Hammerstein equations via an interpolation method," *Appl. Math. and Comp.*, vol. 168, pp. 141–145 2005.
- [12] K. Maleknejad and M. Rabbani, "Using the Petrov-Galerkin elements for solving Hammerstein integral equations," *Appl. Math. and Comp.*, vol. 172, pp. 831–845 2006.
- [13] K. Maleknejad and H. Derili, "Numerical solution of Hammerstein integral equations by using combination of spline-collocation method and Lagrange interpolation," *Appl. Math. and Comp.*, vol. 190, pp. 1557–1562 2007.
- [14] K. Maleknejad and H. Derili, "The collocation method for Hammerstein equations by Daubechies wavelets," *Appl. Math. and Comp.*, vol. 172, pp. 846–864, 2006.
- [15] F. Stenger, *Numerical methods based on sinc and analytic functions*, Springer-Verlag, New York, 1993.
- [16] M. Muhammad and M. Mori, "Double exponential formulas for numerical indefinite integration," *Journal of Computational and Applied Mathematics*, vol. 161, pp. 431–448, 2003.
- [17] K. E. Atkinson and H. Weimin, *Theoretical numerical analysis, A Functional Analysis Framework*, 2nd ed., Springer-Verlag, 2000.
- [18] J. Stoer and R. Bulirsch, *Introduction to numerical analysis*, 3rd ed., New York: Springer-Verlag, 2002.
- [19] H. R. Marzban, H. R. Tabrizidooz and M. Razzaghi, "A composite collocation method for the nonlinear mixed Volterra-Fredholm-Hammerstein integral equations," *Commun Nonlinear Sci Numer Simulat*, vol. 16, pp. 1186–1194, 2011.
- [20] N. Madbouly, "Solutions of Hammerstein integral equations arising from chemical reactor theory," Ph.D. thesis, University of Strathclyde, 1996.

Enumerating Order 7 de Bruijn Sequences

Gregory L. Mayhew

Adjunct Professor of Electrical Engineering
Department of Electrical and Systems Engineering
Washington University St. Louis
One Brookings Drive, Urbauer 211
St. Louis, MO 63130-1127

Abstract—Order n de Bruijn sequences are the period 2^n binary sequences from n -stage feedback shift registers. The de Bruijn sequences have randomness properties that are useful in data link security. Many de Bruijn sequences have nearly ideal two-level autocorrelation properties necessary for signal acquisition. The nonlinear generators of de Bruijn sequences produce forward error correction codes which achieve the maximum free distance as specified by Heller-Griesmer bound. This paper provides new results on weight class properties of de Bruijn sequences. These properties were investigated using numerical algorithms with parallel processing on a 64 node Evolocivity II cluster^{1,2}.

TABLE OF CONTENTS

1. INTRODUCTION
2. WEIGHT CLASSES
3. INVESTIGATIVE METHODOLOGY
4. ORDER 7 DE BRUIJN DATA
5. CONCLUSIONS

1. INTRODUCTION

Algebraically constructed binary sequences with randomness properties have many applications in logic synthesis, coding theory, cryptography, and communications. The order n de Bruijn sequences are the period 2^n binary sequences generated recursively using nonlinear n -stage feedback shift registers. The de Bruijn sequences exhibit the balance, run, and span- n randomness properties [1]. Also, the de Bruijn sequences have linear spans greater than half the sequence length [2]. The de Bruijn sequences can be cataloged by the Hamming weight of their generating functions [3]. This paper presents new information on the weight class distribution of order 7 de Bruijn sequences.

2. WEIGHT CLASSES

The de Bruijn sequences have generator functions of the form $x_n \oplus g(x_{n-1} \dots x_2 x_1)$. The *weight* w is the number of

logical ones (Hamming weight) in the 2^{n-1} entries in the truth table $g(x_{n-1} \dots x_2 x_1)$. Truth tables producing the $2^{2^{n-1}-n}$ de Bruijn sequences have odd weights between the minimum weight $Z(n)-1$ and the maximum weight $2^{n-1}-Z^*(n)+1$, inclusive [3]. $Z(n)$ is the number of cycles from the pure cycling register whose truth table $g(x_{n-1} \dots x_2 x_1)$ has all-zeros. Making $Z(n)-1$ changes in the PCR truth table can merge the cycles to form a de Bruijn sequence.

$$Z(n) = \frac{1}{n} \sum_{\substack{d|n \\ \text{all:d}}} \phi(d) 2^{n/d}$$

$Z^*(n)$ is number of cycles from the complementing cycling register whose truth table $g(x_{n-1} \dots x_2 x_1)$ has all-ones. Making $Z^*(n)-1$ changes in the CCR truth table can merge the cycles to form a de Bruijn sequence.

$$Z^*(n) = \frac{1}{2n} \sum_{\substack{d|n \\ \text{odd:d}}} \phi(d) 2^{n/d}$$

$\phi(d)$ is the Euler totient function.

In order 7, there are 144,115,188,075,855,872 de Bruijn sequences. They are distributed among the 19 weight classes having odd weights between 19 and 55, inclusive.

3. INVESTIGATIVE METHODOLOGY

The number of candidate truth tables in each weight class is the number of ways to put $w-2$ logical ones in $2^{n-1}-2$ entries of truth table $g(x_{n-1} \dots x_2 x_1)$. Let $C(h, r)$ denote binomial coefficient. States zero and $2^{n-1}-1$ of $g(x_{n-1} \dots x_2 x_1)$ are always a logical one, so the number of candidates for a weight class is the binomial coefficient $C(2^{n-1}-2, w-2)$. Table I shows the number of cases, which will be mitigated by considering the impact of subcycles (short cycles).

¹ 2011 Parallel and Distributed Processing Techniques and Applications

² PDP 2628

Table I - Cases in Order 7 Weight Classes

Weight Class	Candidate Truth Tables C(62,w-2)
19	739,632,519,584,070
21	4,282,083,008,118,300
23	18,412,956,934,908,690
25	59,678,358,445,158,600
27	147,405,545,359,541,742
29	279,692,573,246,309,972
31	409,894,288,378,212,890
33	465,428,353,255,261,088
35	409,894,288,378,212,890
37	279,692,573,246,309,972
39	147,405,545,359,541,742
41	59,678,358,445,158,600
43	18,412,956,934,908,690
45	4,282,083,008,118,300
47	739,632,519,584,070
49	93,052,749,919,920
51	8,308,281,242,850
53	508,271,323,092
55	20,286,591,270
Total	2,305,741,118,628,006,748

If an odd weight truth table and starting state are picked at random, then the length of resulting sequence is modeled as a uniform probability between n and 2^n [6]. In order 7, only 1 in 16 truth tables produce a de Bruijn sequence.

Consider an invalid truth table candidate that generates three subcycles with lengths 3, 5, and 2^n-8 . In this instance, odds

favor that a state belonging to the subcycle length 2^n-8 will be picked. Iteratively applying the feedback function will build a sequence with length 2^n-8 . So odds favor constructing a maximum length subcycle before ascertaining that the chosen truth table for $x_n \oplus g(x_{n-1} \dots x_2 x_1)$ will not yield a de Bruijn sequence. A better method is to discover the shortest subcycles first and rapidly discard any invalid functions.

These improvements are analogous to primality testing of integers in number theory. Highly composite integers are generally easy to discover because their primes are usually small. Truth tables yielding factored or “composite” de Bruijn graphs with small subcycles can be identified as easily as small prime factors are identified in integers.

The canonical forms of the length k subcycles are the cyclic equivalency classes (CEC) in a Bounded Synchronization Code [1]. The number of binary CECs is

$$\beta(k, k) = \frac{1}{k} \sum_{d|k} \mu(d) 2^{k/d} = \frac{1}{k} \sum_{d|k} \mu\left(\frac{k}{d}\right) 2^k$$

For each length k cycle, there exists some shift register length m that is the shortest shift register which can produce this particular length k cycle and guarantee each state has a unique successor. This value m is the minimum span of the length k cycle, called an $\{m,k\}$ sequence [3].

The appropriate subcycle tests for the order n truth table candidates are $\{m,k\}$ sequences with $m \leq n$. Table II lists the lexicographically least member of each CEC for $k \leq 5$ and gives conditions for the existence of these subcycles in an odd weight feedback function $g(x_6 \dots x_2 x_1)$ in order $n = 7$.

Table II - Length $k \leq 5$ Cycles in Order $n = 7$ Functions

Cycle length k	CEC based Subcycle	Existence Test in $g(x_6 x_5 x_4 x_3 x_2 x_1)$
2	01	$g(21)=1, g(42)=1$
3	001	$g(9)=1, g(18)=0, g(36)=1$
	011	$g(27)=1, g(45)=0, g(54)=1$
4	0011	$g(12)=0, g(25)=1, g(38)=1, g(51)=0$
	0001	$g(4)=1, g(8)=1, g(17)=0, g(34)=0$
	0111	$g(29)=0, g(46)=0, g(55)=1, g(59)=1$
5	00011	$g(6)=1, g(12)=0, g(24)=1, g(35)=1, g(49)=1$
	00111	$g(14)=1, g(28)=1, g(39)=1, g(51)=0, g(57)=1$
	00001	$g(2)=1, g(4)=0, g(8)=0, g(16)=1, g(33)=0$
	00101	$g(10)=1, g(18)=0, g(20)=1, g(37)=0, g(41)=0$
	01011	$g(22)=0, g(26)=0, g(43)=1, g(45)=0, g(53)=1$
	01111	$g(30)=0, g(47)=1, g(55)=0, g(59)=0, g(61)=1$

For each investigation of a particular weight class w , every weight $w-2$ combinatorial pattern for $g(x_{n-1} \dots x_2 x_1)$ is built. Each pattern is progressively examined for subcycles from length two to length five. Once a subcycle is discovered, the pattern is discarded and the next pattern is built. If a pattern passes subcycle testing, then this pattern is used as a feedback function to build a sequence. Whenever a starting state reoccurs, the sequence length is measured. Sequences are tallied if a period 2^n de Bruijn sequence was produced.

4. ORDER 7 DE BRUIJN DATA

Table III incorporates the new data for weight classes 21, 45, and 47 with existing data on the weight class distribution for order 7 de Bruijn sequences. The number of sequences in each weight class is expressed concisely as a coefficient with its symmetry group [4]. Each order 7 de Bruijn weight class contains a symmetry group of 2^{26} sequences.

Table III - Weight Class Data for Order 7 de Bruijn Sequences

Weight Class	de Bruijn Sequences	Coefficient's Prime Decomposition
19	$91,125 * 2^{26}$	$3^6 * 5^3$
21	$1,458,000 * 2^{26}$	$2^4 * 3^6 * 5^3$
23 to 43	not known	
45	$3,041,872 * 2^{26}$	$2^4 * 41 * 4,637$
47	$476,300 * 2^{26}$	$2^2 * 5^2 * 11 * 433$
49	$52,240 * 2^{26}$	$2^4 * 5 * 653$
51	$3,773 * 2^{26}$	$7^3 * 11$
53	$160 * 2^{26}$	$2^5 * 5$
55	$3 * 2^{26}$	3

The weight class 23 investigation is 37 percent complete. At this time, the coefficient estimate is 10,576,243.

Unlike previous orders, in order 7 the number of sequences in every weight class is not divisible by the number of sequences in the maximum weight class.

The order 7 investigations were performed using a 64 node Linux Network Evolocity II cluster. Each node contained dual Pentium Xeon processors operating at 2.4 GHz with 512 Megabyte L2 cache, 2 Gigabyte ECC PC2100 DDR SRAM, and Red Hat Enterprise Linux 4 operating system. The numerical algorithms were written in C programming language. Weight class 47 required 67,241 CPU hours over a one year period. Weight class 21 required 394,662 CPU hours over two years. Similarly, weight class 45 required 403,209 CPU hours over another two years.

5. CONCLUSIONS

The coefficients' prime decompositions do not have any apparent relationships from number theory. Rather, dividing the number of sequences in each weight class by the total number of de Bruijn sequences yields the order n probability mass function for the de Bruijn sequences per weight class. The product of a Gaussian and a Beta distribution accurately models the order n probability mass function of de Bruijn sequences per weight class [5, 6].

ACKNOWLEDGEMENT

My special thanks to Ulysses Okawa for his insight on several programming techniques.

REFERENCES

- [1] S. W. Golomb, Shift Register Sequences, Aegean Park Press, Laguna Hills, CA 2nd Ed., 1982
- [2] T. Etzion and A. Lempel, "On the distribution of de Bruijn sequences of given complexity", IEEE Transactions Information Theory 30, 1984, pp. 611-614.
- [3] H. Fredricksen, "A survey of full length nonlinear shift register cycle algorithms", SIAM Review 24, 1982, pp. 195-229.
- [4] E. R. Hauge and J. Mykkeltveit, "The analysis of de Bruijn sequences of non-extremal weight", Discrete Mathematics, Vol. 189, 1998, pp. 133-147.
- [5] G. L. Mayhew, "Probability Mass Functions for de Bruijn Weight Classes", Proceedings IEEE Aerospace Conference, March 2006, Session 11, Track 4, pp. 1-12.
- [6] G. L. Mayhew, "Cumulative Distribution Function for Order 7 de Bruijn Weight Classes", Proceedings IEEE Aerospace Conf., March 2009, Session 8, Track 9, pp. 1-9.

BIOGRAPHY

Gregory Mayhew has been continuously involved in the design, analysis, development, and operational testing of secure, spread spectrum communication networks on a variety of projects at Hughes Electronics and Boeing Phantom Works. He is currently developing communication architectures in network centric operations using MANET. Gregory has a bachelor's degree in electronics from the Massachusetts Institute of Technology, a master's degree in digital communications from the University of Southern California, and a doctorate in information theory from the University of Southern California. Gregory is IEEE Senior Member, Boeing Technical Fellow, and Adjunct Professor of Electrical Engineering at Washington University St Louis.

SESSION
GRID AND CLOUD COMPUTING

Chair(s)

TBA

FTPProfiler: A New Profiling Tool for GridFTP Servers

Huong Luu¹, Rajkumar Kettimuthu^{2,3}, Marianne Winslett¹

¹ Department of Computer Science, University of Illinois at Urbana-Champaign, USA

² Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, USA

³ Computation Institute, University of Chicago/Argonne National Laboratory, Chicago, USA

Abstract— GridFTP is a high-performance, secure, and reliable data transfer protocol that is being widely used in data transmission in Grid computing. A GridFTP server needs to achieve high throughput as it sends and receives data to and from multiple sources, each with its own configuration. Profiling tools can potentially help GridFTP administrators gain insight into the system's activities and identify configuration tradeoffs as well as potential bottlenecks. This paper presents a new profiling tool, called FTPProfiler, which is built upon standard system profiling tools OProfile and Sar, to provide a more complete view of the system and simplify the profiling process for new GridFTP servers. FTPProfiler calls OProfile and Sar, analyzes their profiling results and generates a detailed report. Through a case study, we show how FTPProfiler can help administrators understand the effects of system parameters such as the TCP buffer size, block size, and parallel TCP streams on server performance and load, thus simplifying the process of detecting bottlenecks and tuning for performance.

Keywords – Profiling GridFTP, Profiling WAN data movement, Profiling high-speed transfers

I. INTRODUCTION

GridFTP [1] extends the standard FTP [2] protocol to provide a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. The Globus GridFTP implementation [3] has been widely used for data transfer in the Grid community. It provides a modular and extensible data transfer system architecture suitable for wide area and high-performance environments. To get the maximum performance from the GridFTP server, a variety of parameters need to be tuned. Profiling tools can potentially help GridFTP administrators gain insight into the system's activities, identify configuration tradeoffs, and understand their impact on server load.

Currently two good profiling tools, *OProfile* [21] and *Sar* [23], can provide information on different aspects of the system behavior without imposing much overhead. In this paper we present FTPProfiler, which leverages *OProfile* and *Sar* to provide a more complete view of the GridFTP system and simplify the profiling process on new servers. Through our case studies, we demonstrate the use of FTPProfiler to profile system, to identify the potential bottleneck and to understand the effects of system parameters on server behavior, including TCP buffer size, I/O block-size, and the number of parallel streams or otherwise called parallelism.

In the remainder of the paper, we provide background on GridFTP in Section 2, discuss the design of FTPProfiler

in Section 3 and present the profiling case studies in Section 4. We describe related work in Section 5 and summarize in Section 6.

II. GRIDFTP

FTP is a widely implemented and well-understood IETF-standard protocol. It provides a well-defined architecture for protocol extensions and supports dynamic discovery of the extensions supported by a particular implementation. Many extensions for FTP have been defined through the IETF. The FTP protocol also separates control and data channels, enabling third-party transfers, that is, the transfer of data between two end hosts, mediated by a third host. The GridFTP protocol is based on FTP and thus also benefits from the FTP protocol advantages mentioned above. In addition, it extends the FTP protocol to provide a high-performance, secure, and reliable data transfer protocol optimized for high-bandwidth wide-area networks.

The following is a summary of key GridFTP features.

Third-party control of data transfer. To manage large datasets for distributed communities, we must provide authenticated third-party control of data transfers between storage servers. A third-party operation allows a user or application at one site to initiate, monitor and control a data transfer operation between two other sites: the source and destination for the data transfer.

Authentication, data integrity, and data confidentiality. GridFTP supports Generic Security Services (GSS)-API authentication of the control channel (RFC 2228) and data channel (GridFTP extensions), and supports user-controlled levels of data integrity and/or confidentiality. Data channel authentication is of particular importance in third party transfers, since the IP address of the host connecting for the data channel will be different than that of the host connected on the control channel, and there must be some way to verify that it is the intended party.

Striped data transfer. Data may be striped or interleaved across multiple servers, as in a parallel file system or DPSS disk cache [19]. Thus, GridFTP defines protocol extensions that support the transfer of data partitioned among multiple servers.

Parallel data transfer. On wide-area links, using multiple TCP streams in parallel between a single source

and destination can improve aggregate bandwidth relative to that achieved by a single stream [17, 20]. GridFTP supports such parallelism via FTP command extensions and data channel extensions. A GridFTP implementation can use long virtual round trip times to achieve fairness when using parallelism or striping [18]. Note that striping and parallelism may be used in tandem, i.e., users may have multiple TCP streams open between each of the multiple servers participating in a striped transfer.

Partial file transfer. Some applications can benefit from transferring portions of files rather than complete files, such as analyses that require access to subsets of massive files. FTP allows transfer of the remainder of a file starting at a specified offset. GridFTP supports requests for arbitrary file regions.

Automatic negotiation of TCP buffer/window sizes. Using optimal settings for TCP buffer/window sizes can dramatically improve data transfer performance. However, manually setting TCP buffer/window sizes is an error-prone process, particularly for non-experts, and is often simply not done. GridFTP extends the FTP command set and data channel protocol to support both manual setting and automatic negotiation of TCP buffer sizes for large files and for large sets of small files.

Support for reliable and restartable data transfer. Reliable transfer is important for many applications that manage data. Fault recovery methods are needed to handle failures such as transient network and server outages. The FTP standard includes basic features for restarting failed transfers, but these are not widely implemented. GridFTP exploits these features and extends them to cover its new data channel protocol.

The Globus implementation of GridFTP provides all these key features along and is highly extensible. Its modular architecture enables a standard GridFTP-compliant client to access any storage system that implements its data storage interface [5], including the HPSS archival storage system [6], SRB [7], the PVFS parallel file system [8], the GPFS parallel file system [9], and POSIX [10] file systems. Its eXtensible I/O interface [11] allows GridFTP to target high-performance wide-area communication protocols such as UDT [12], FAST TCP [13], and RBUDP [14]. Globus GridFTP is optimized to handle a variety of types of datasets, from a single, huge file to datasets comprising lots of small files [15, 16].

III. FTPROFILER OVERVIEW

In general, GridFTP server performance profiling needs to be done in a way that minimizes interference with the production jobs running on the server. Thus the steps of preparation and generating reports are usually done on a separate machine, rather than the server itself. For this

reason, FTPProfiler runs on the client machine where the transfer is initiated for a client-server transfer or a third party transfer between servers. During execution, FTPProfiler first remotely accesses the server to start the underlying tools, including *OProfile* and *Sar*, then runs the test workload on the client machine. After finishing the run, FTPProfiler again accesses the server to turn off the running profiling tools and post-process the results, which are then sent back to the client machine. Based on the results received, FTPProfiler generates a report that covers the essential information. The user has the option of viewing additional information from the profiling result files. The working scenario of FTPProfiler is shown in Figure 1.

FTPProfiler uses *OProfile* and *Sar* as underlying tools because they provide different aspects of system profiling with low overhead. *OProfile* uses the hardware performance counters of the CPU to profile the entire system. *OProfile* shows how time is spent in different parts of the system, such as the kernel, kernel modules, interrupt handlers, shared libraries and applications. On the other hand, *Sar* is capable of profiling the memory usage, including memory and swap space utilization and other performance metrics for each processor. FTPProfiler processes the results from both tools to provide essential information to the user, such as CPU utilization, peak and average load.

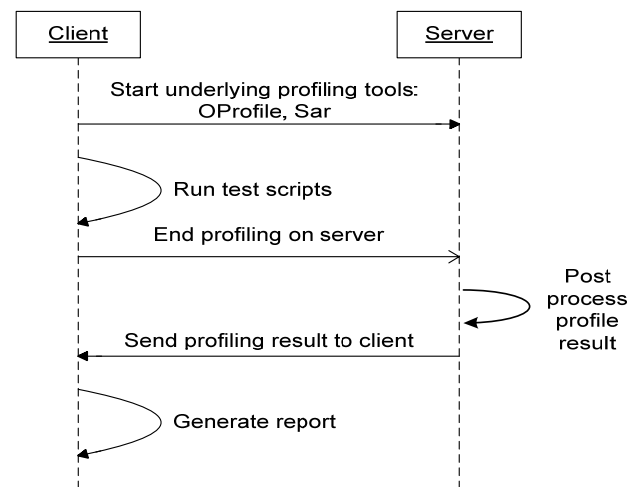


Figure 1: Working scenario for FTPProfiler

IV. CASE STUDY

The goal of our experiments was to perform a performance study to see the effects of different parameters to GridFTP performance. In this process, we demonstrate the use of FTPProfiler and other tools to detect the bottleneck of the transfers. To this end, we used five servers in multiple locations, to model both local and distant data transfers. For the local case, the client and server are nodes in the Breadboard cluster at Argonne National Laboratory, each with 4GB of main memory,

approximately 300GB of storage per node, connected by a Myri-10G 10Gbps network with a 1Gb/s network interface card (NIC) on each machine.

For the distant case, we used dedicated GridFTP servers at different TeraGrid sites, namely the servers Pople at the Pittsburgh Supercomputing Center, Abe at the National Center for Supercomputing Applications, and Ranger at the Texas Advanced Computing Center. These servers can utilize the TeraGrid backbone network (10 – 30 Gb/sec) to the fullest for a better transfer rate. However, Pople uses 1 Gb/s NIC while Abe and Ranger have 10 Gb/s NICs.

We had the privileges to adjust server configurations on Breadboard nodes, but not elsewhere. Our experiments tune the most common options for performance: TCP buffer size, parallel TCP streams, and buffer size. We transfer 16GB of data with files of size 800MB, 1GB, and 4GB.

A. TCP buffer size

The TCP buffer size option (*-tcp-bs*) specifies the size of the TCP buffer to be used by the underlying FTP data channels. This parameter has long been believed to be critical to achieve good performance over the WAN. The optimal value for the TCP buffer size is believed to be the Bandwidth * Delay Product (BDP), which depends on the available Bandwidth and Round Trip Time (RTT) between two destinations. However, in most modern operating systems, TCP buffers are automatically tuned which means the buffer size is automatically adjusted based on the changing network conditions. This raises the question: will manually changing the TCP buffer size hurt or help, in the presence of auto-tuning? We investigated this question through a performance study of the effect of TCP buffer size parameter on GridFTP transfers.

For the case where the RTT between the client and the server is on the order of few milliseconds or less, the TCP buffer size does not affect the transfer rate much, hence it will not be the source of a bottleneck. Thus we tested the effect of this parameter in the long distance case. We transferred data between TeraGrid sites and the server on the Breadboard cluster. We tested with different TCP buffer sizes: the default option of GridFTP (which uses autotuning), a manually calculated BDP value and fixed values of 4MB, 8MB, 16MB and 32MB. The 4MB, 8MB, 16MB and 32MB values are recommended for the optimal transfer rate among TeraGrid servers, and are actually used by TeraGrid for this purpose. In all tests we have conducted, parameters other than the one being tested are set to the default values.

To calculate BDP, we used Iperf [24] to measure bandwidth and ping to determine the RTT between sites. To assess GridFTP servers' performance, we measured the transfer rate of each configuration multiple times, eliminated any outliers, then calculated the average value.

The transfers were performed at different times of day to take into account the effects of loads created by other users in a shared environment like TeraGrid. Based on our measurements, the BDP values are approximately 500KB (Abe and BB), 2MB (Pople and Abe, Pople and BB), 3MB (Ranger and Abe, Ranger and Pople) and 4.5MB (Ranger and Pople).

One disadvantage of using BDP rather than other options is that BDP needs to be calculated for every connection that we want to improve, while auto-tuning and fixed buffer sizes do not require that. Manually choosing the TCP buffer size might worsen the performance because too-large buffers possibly overload the receiver's TCP window and create congestion on the server. This is a serious problem called the "buffer bloat" problem [28, 29], discussed by Jim Gettys, in which excessively large buffers in the network communication system eventually lead to network congestion, destroy congestion avoidance in the transport protocols and cause poor performance.

On the other hand, sometimes the TCP settings are too small, thus limiting the server's performance and under-utilizing the network. Hence manual tuning is generally not recommended over auto-tuning.

As shown in Figure 2, with auto-tuning, the choice of TCP buffer size is not as important as it was in the past. Among TeraGrid servers, the difference between the best and the worst transfer rates is about 10% except for the Ranger to Pople case, where the default auto-tuning gives a 46% improvement compared to the worst buffer size, 16MB. The Breadboard server worked best with the auto-tuning option and the performance quickly dropped when TCP buffer size increased which might indicate the bottleneck in TCP settings. Figure 2 also shows that the transfer rates in different directions for the Ranger server are not the same; in fact, the rate from Abe to Ranger is double the reverse direction. This could mean that Ranger has different network configurations for each direction. Ranger seems to prioritize its configuration for incoming traffic over outgoing.

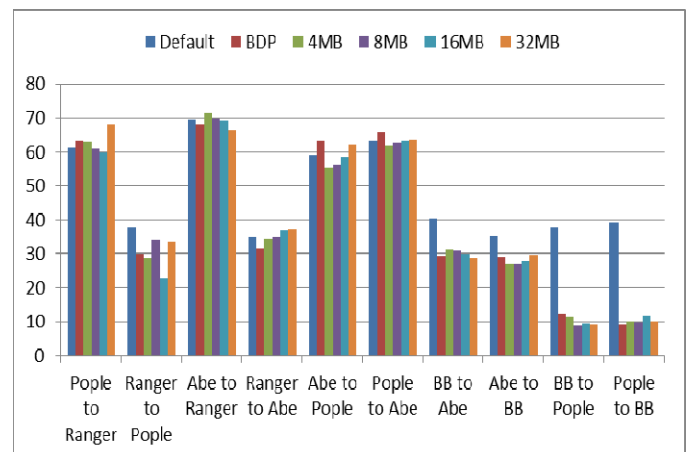


Figure 2: Transfer rates (MB/s) with different TCP buffer size values

	With old TCP settings				With new TCP settings			
	Peak load	Transfers to I/O devices per second	Network packets received per second	Network packets transmitted per second	Peak load	Transfers to I/O devices per second	Network packets received per second	Network packets transmitted per second
Default	23.7%	4.85	11675	15529	34.9%	9.43	45372	61977
BDP	20.6%	2.11	5734	7664	20.4%	10.78	38332	52539
4MB	20.5%	2.87	7271	9673	28.2%	11.63	38725	53191
8MB	20.6%	3.40	8352	11083	34.5%	12.00	39021	53618
16MB	21.4%	3.99	9524	12635	34.5%	10.51	33798	46433
32MB	20.1%	4.49	10527	13947	34.0%	10.92	34483	47385

Table 1: GridFTP server profiling result using different TCP settings

To explain the low performance of transfers between Breadboard and Abe/People in Figure 2, we used FTPProfiler to gain more insight into this problem. The profiling result for the Breadboard server, shown in the left half of Table 1, suggested that the current TCP settings under-utilize the server with a very low transfer rate to the physical I/O device (server) as well as the network transfer rate. This indicates that network setting could potentially be the bottleneck in this case.

To understand the impact of TCP settings, we examined the settings on each server; the main differences between servers' settings are listed in Table 2. As we can see, the TCP settings in Breadboard are quite small and hence, the transfer rate from/to Breadboard server is low because of this limitation. It also explains why Breadboard server performance decreased as we increased the TCP buffer size, as shown in Figure 2.

We adjusted the settings of the Breadboard server to improve its performance, using the sysctl command [25]. We changed the Breadboard server auto-tuning settings to be the values suggested in TCP tuning manuals [22]. The suggested values are very close to Abe's except that `netdev_max_backlog` becomes 30,000 instead of 250,000. After changing the TCP settings, the transfer rates using different buffer sizes all improved greatly, as much as five times faster, as shown in Figure 3. Further, once the TCP settings were set appropriately, the effect of the buffer size is small. The Breadboard profiling result with new settings, shown in the right half of Table 1, also confirmed this.

In conclusion, auto-tuning does a good job in achieving good performance without having to manually select a value for the TCP buffer size. However, we might need to adjust other TCP settings to make sure that we can fully take advantage of auto-tuning.

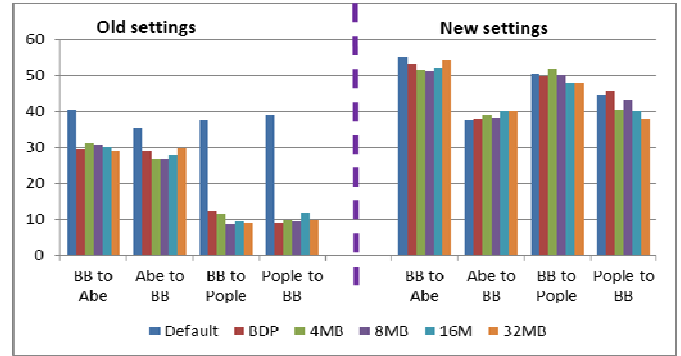


Figure 3: Transfer rate (MB/s) for different TCP buffer sizes, after adjusting Breadboard's other TCP settings

B. Parallelism

This parameter specifies the number of TCP streams running in parallel to be used in the transfer. It is one of the most commonly tuned parameters to achieve good performance because it improves the aggregate bandwidth and makes better use of the available bandwidth. The default value for parallelism is 1. In these tests, we auto-tune the TCP buffer size and use the new settings for Breadboard's other TCP parameters, as described in the previous section. Other parameters are set to default.

The experiment results are presented in Figure 4. In general, using parallel streams improves the transfer rate quite significantly, especially when the RTT between servers is large. For Abe and Breadboard transfers whose RTT is only 6.5 ms, increasing the number of parallel streams does not help. It is interesting to observe that even though Ranger does not have a good outgoing rate using the TCP buffer size option, we can actually improve it significantly with the parallelism option. And the transfer from Abe to People requires further investigation, but our

Variable: meaning	Abe	People	Ranger	Breadboard
<code>net.core.rmem_max/wmem_max</code> : Set max size of TCP receive/transmit window.	16MB	2MB	32MB	128KB
<code>net.core.rmem_default/wmem_default</code> : Set default size of TCP receive/transmit window.	112KB	256KB	64KB	122KB
<code>net.ipv4.tcp_rmem</code> : Set min, default, max receive window.	4KB/ 85KB/ 16MB	4KB/ 85KB/ 16MB	4KB/ 85KB/ 32MB	4KB/ 85KB/ 4MB
<code>net.ipv4.tcp_wmem</code> : Set min, default, max transmit window.	4KB/ 64KB/ 16MB	4KB/ 16KB/ 256KB	4KB/ 12MB/ 32MB	4KB/ 16KB/ 4MB
<code>net.ipv4.tcp_mem</code> : Set min, default, max allocatable TCP buffer space.	1.5MB/ 2MB/ 3MB	48KB/ 64KB/ 96KB	768KB/ 1MB/ 32MB	367KB/489KB/734KB
<code>net.core.netdev_max_backlog</code> : Maximum number of packets in the receiver's queue.	250,000	2500	400,000	1000

Table 2: Servers' TCP settings

privileges only allow in-depth investigation on Breadboard.

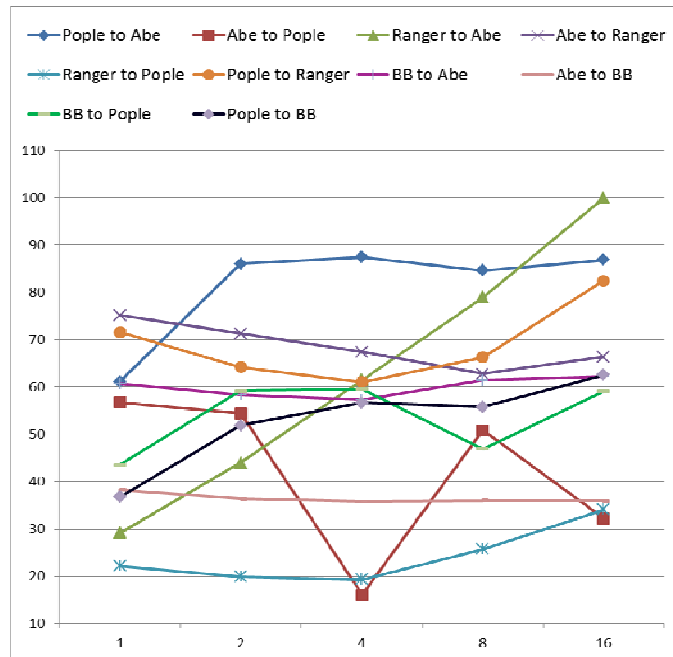


Figure 4: Throughput (MB/s) with 1-16 parallel TCP streams

For transfers from Breadboard to Pople, increasing parallelism (P) significantly helped to improve the transfer rate for small P, but this trend reversed for P=8. By looking at the profiling result shown in Table 3, we see that P=4 improves total throughput without increasing kernel load. So for this transfer, we could recommend to use P=4 to maximize the server performance.

	P=1	P=2	P=4	P=8	P=16
Transfer rate (MB/s)	45.86	58.36	59.06	47.01	58.33
Peak CPU utilization (%)	22.2	39.1	41.7	38.7	36.4
Kernel load (%)	45.22	48.60	46.73	53.94	59.26

Table 3: Server profiling for Breadboard to Pople transfer case

C. Block size:

This parameter specifies the size of the buffer that the underlying IO system uses when posting read requests to the disk. This parameter gives the user more control, as each underlying IO system that GridFTP uses has its own optimal IO buffer size value. As this parameter is only applicable to read requests, it affects only the data sender. We must distinguish between two cases – sending from the client or server – to make sure we adjust the right value.

When the client sends data to the server, we specify the block size value using the `-bs` option in `globus-url-copy` command. In the third party transfer case, in which a client initiates the transfer between two servers or when the server sends data to the client, we adjust the server's configuration by adding the block size option when starting the server. The default value is 256KB. Other parameters are set to default.

As shown in Figure 5, the experiments' result shows that the adjustment does not significantly improve the performance. In fact the default value for block size (256KB) performs slightly better overall than other values.

Intuitively, as long as the block size is big enough to keep the rest of the system fed with data, it will not be the bottleneck. We, therefore suspect that it might become the bottleneck if there is more load, for example with 16 parallel streams. However, the impact of increasing the block size is still small, for 16-way parallel transfers from a Breadboard client to the Abe server.

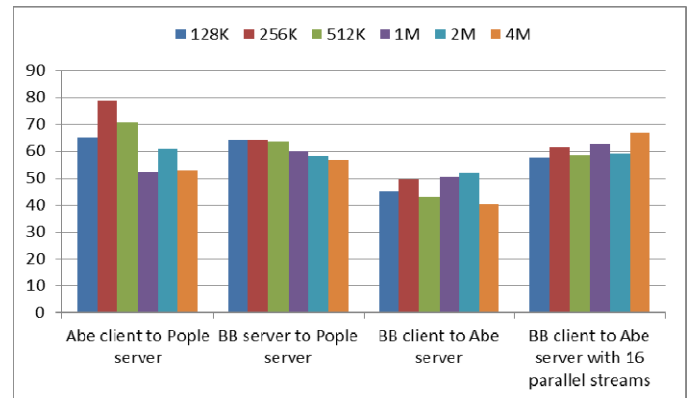


Figure 5: Effects of adjusting block size

D. Putting everything together: How well do we do?

In general, there are two types of GridFTP transfer: one that involves disk activity and another one that does not, which is memory-to-memory transfer. The latter type is usually used to identify if the performance bottleneck lies in the network configuration or in disk I/O. In this section, we first perform memory-to-memory transfers between servers and then by comparing the memory-to-memory transfer rate to the best disk-to-disk transfer rate, we find out how efficiently we have utilized the network for disk related transfer.

	Abe	Breadboard	Pople	Ranger
Abe		110.9	113.9	552.95
Breadboard	111		105.98	111.33
Pople	115.94	84.3		115.32
Ranger	434.47	NA	111.78	

Table 4: Memory to memory transfers between servers, in MB/s

As shown in Table 4, memory-to-memory transfers from or to Breadboard and Pople were able to saturate the network and reach close to the bandwidth limit (bounded by 1 Gb/s NIC (approximately 125 MB/s)). Abe and Ranger servers have 10 Gb/s NIC. That's why the transfer rates between Abe and Ranger are 550MB/s and 434MB/s. Obviously, we were not able to saturate the network link in this case. The throughput is only 35-45% of the available capacity. It could be due to network bottlenecks or because the end systems are not powerful enough to drive a 10Gb/s network link. The transfer rate from Pople to Breadboard

was about two thirds of the NIC limit, which might indicate bottlenecks in network configuration.

Next, we compared the best achievable disk-to-disk transfer rate (using all performance-improving parameter settings discussed so far) to the memory-to-memory transfer rate, to identify the disk I/O overhead. Our preliminary results in Table 5 indicated that we were not using the network efficiently. FTPProfiler can help us gain insight into where the bottleneck might be and improve the performance optimization.

	Best disk to disk transfer rate (MB/s)	Memory to memory transfer rate	Percentage utilization
Abe to BB	40.32	110.90	36%
Abe to Pople	56.76	113.90	50%
Abe to Ranger	75.13	552.95	14%
BB to Abe	62.25	111.00	56%
BB to Pople	59.53	105.98	56%
Pople to Abe	86.82	115.94	75%
Pople to BB	45.41	84.30	54%
Pople to Ranger	82.41	115.32	71%
Ranger to Abe	99.99	434.47	23%
Ranger to Pople	37.84	111.78	34%

Table 5: Percentage of network utilization for best disk to disk transfers

We first measured the GridFTP transfer rate from Breadboard clients to the Breadboard server. We measured the local disk-to-disk transfer because on the Breadboard system we have exclusive access to the nodes used in the test, so that the transfer rate is not affected by the load created by other clients. The memory to memory transfer rate was 112MB/s. However, the disk-to-disk transfer rate between Breadboard nodes is only about 60MB/s. We also measured the memory to disk and disk to memory transfer rates, which were 50MB/s and 80MB/s, respectively.

We expected the transfer rate to be close to the rate measured by a standard disk benchmark such as FIO [26] or Bonnie [27]. We used Bonnie to measure Breadboard nodes' disk performance, focusing on block sequential IO. However, the rate reported by Bonnie is 100MB/s and 91MB/s for output and input respectively, which is much higher than the GridFTP disk IO performance.

To understand where the bottleneck might be, we profiled both source and destination server for all transfer cases between memory and disk. The results are shown in Table 6. In this table, `/mbcache` refers to the filesystem meta information block cache. `/nfs` is the file system module, `/sunrpc` is the protocol for making remote procedure calls. `/tcp_cubic` module provides the cubic congestion control protocol and `/tg3` module is the Ethernet NIC driver.

We discovered several interesting patterns. First, `/nfs` is heavier on the destination server with disk I/O because it has to allocate disk blocks to write data to. `/mbcache` is disk I/O related only; `/sunrpc` runs mainly on the source server with disk-related transfer while `/tcp_cubic` only appears on

the source server, and `/tg3` is needed in all cases. This observation suggests that we should focus on improving these related modules for disk transfer cases.

	/mbcache	/nfs	/sunrpc	/tcp_cubic	/tg3
Mem-to-mem: Log on source	0	0	0	0.148	2.403
Mem-to-mem: Log on destination	0	0	0	0	3.389
Mem-to-disk: Log on source	0	0	0	0.087	2.008
Mem-to-disk: Log on destination	0.004	2.423	0.364	0	2.615
Disk-to-mem: Log on source	0.005	0.428	1.048	0.014	2.341
Disk-to-mem: Log on destination	0	0	0	0	2.943
Disk-to-disk: Log on source	0.006	0.412	1.209	0.038	2.703
Disk-to-disk: Log on destination	0.005	2.395	0.240	0	2.387

Table 6: Percentage of load on server for important modules

V. RELATED WORK

The work we present in this paper is similar to that of George Kola et al. [4], who performed a full system profiling and comparison between GridFTP and NeST servers, the very commonly used data servers at that time. Based on the profiling result, they discussed the configuration tradeoffs of parallel streams and the block-size parameter for server performance and server load. Their experiments were carried out with GridFTP 2.4.3. Since then, Globus Alliance has released several other versions of GridFTP with many improvements and new features. Our experiments are performed with Globus Toolkit 5.0.0 and GridFTP server 3.19, and thus provide a more current view of GridFTP.

VI. SUMMARY AND FUTURE WORK

We have developed FTPProfiler, a profiling tool for GridFTP. This tool was motivated by the need to understand the effects of various system parameters in performance tuning and detecting bottlenecks. We have demonstrated the use of this tool using a variety of GridFTP transfers both in the LAN and WAN settings. We have shown the effect of tuning parameters such as TCP buffer size, parallel streams and block size on the performance of GridFTP.

Our performance study has shown that the auto-tuning feature in modern operating systems is doing a good job in adjusting the TCP buffer size automatically based on the changing network conditions. However, to be effective, auto-tuning requires good settings for other TCP parameters. After changing the other TCP settings of the Breadboard server, data transfer rates with auto-tuning were as much as five times faster. We also found that using

parallel streams helps to improve the transfer rate while block size does not show a clear effect on the performance.

Further, we have compared the performance of disk-to-disk transfers with memory-to-memory transfers and identified some bottlenecks in the GridFTP system, using the detailed analysis provided by the FTPProfiler tool.

In future, we plan to extend our study to profile striped servers and analyze the impact of small file optimizations such as pipelining, parallel transfers, and on-the-fly tarring of files. We also intend to do a detailed study of the pros and cons of using TCP versus UDT in WAN settings, using FTPProfiler.

This work was supported in part by NSF grant CCF 0938064 and the Google Summer of Code program.

REFERENCES

- Allcock, W. GridFTP: Protocol Extensions to FTP for the Grid. Global Grid Forum GRD-R-R.020, 2003.
- Postel, J. and Reynolds, J. File Transfer Protocol. Internet Engineering Task Force, RFC 959, 1985.
- W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus Striped GridFTP Framework and Server," SC'05, ACM Press, 2005
- Kola, G., Kosar, T., Livny, M., "Profiling Grid Data Transfer Protocols and Servers", In Proceedings of 10th European Conference on Parallel Processing (EuroPar 2004)
- Kettimuthu, R., Link, M., Bresnahan, J., Allcock, W., "Globus Data Storage Interface (DSI) - Enabling Easy Access to Grid Datasets," 1st DIALOGUE Workshop: Applications-Driven Issues in Data Grids, Aug. 2005.
- Watson, R.W. and Coyne, R.A. The Parallel I/O Architecture of the High-Performance Storage System (HPSS). IEEE MSS Symposium, 1995.
- Baru, C., Moore, R., Rajasekar, A. and Wan, M., The SDSC Storage Resource Broker. 8th Annual IBM Centers for Advanced Studies Conference, Toronto, Canada, 1998.
- Carns, H., Ligon III, W.B., Ross, R.B., and Thakur, R., "PVFS: A Parallel File System For Linux Clusters", Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, October 2000
- General Parallel File System (GPFS), 2004. www-1.ibm.com/servers/eserver/clusters/software/gpfs.html.
- POSIX 1003.1e draft specification "http://www.suse.de/~agruen/acl/posix/posix_1003.1e-990310.pdf"
- Allcock, W., Bresnahan, J., Kettimuthu, R. and Link, J., The Globus eXtensible Input/Output System (XIO): A Protocol-Independent I/O System for the Grid. Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models held in conjunction with International Parallel and Distributed Processing Symposium, 2005.
- Gu, Y. and Grossman, R.L., UDT: An Application Level Transport Protocol for Grid Computing. Second International Workshop on Protocols for Fast Long-Distance Networks, 2003.
- Jin, C., Wei, D.X. and Low, S.H., FAST TCP: motivation, architecture, algorithms, performance. IEEE Infocom, 2004.
- He, E., Leigh, J., Yu, O. and DeFanti, T.A., Reliable Blast UDP: Predictable High Performance Bulk Data Transfer. IEEE Cluster Computing, 2002.
- Bresnahan, J., Link, M., Kettimuthu, R., Fraser, D., and Foster, I., "GridFTP Pipelining," in Teragrid 2007 Conference, Madison, WI, 2007.
- Kettimuthu, R., Sim, A., Gunter, D. Allcock, W., Bremer, P., Bresnahan, J., Cherry, A., Childers, L., Dart, E., Foster, I., Harms, K., Hick, J., Lee, J., Link, M., Long, J., Miller, K., Natarajan, V., Pascucci, V., Raffenetti, N., Ressler, D., Williams, D., Wilson, L., Winkler, L., "Lessons learned from moving Earth System Grid data sets over a 20 Gbps widearea network", 19th ACM International Symposium on High Performance Distributed Computing (HPDC), 2010
- Hacker, T., Athey, B. and Noble, B., The end-to-end performance effects of parallel tcp sockets on a lossy wide-area network. 16th IEEECS /ACM International Parallel and Distributed Processing Symposium, 2002.
- Hacker, T.J., Noble, B.D. and Athey, B.D., Improving Throughput and Maintaining Fairness using Parallel TCP. IEEE InfoCom, 2004.
- Johnston, W., Greiman, W., Hoo, G., Lee, J., Tierney, B., Tull, C. and Olson, D., High-Speed Distributed Data Handling for On-Line Instrumentation Systems. ACM/IEEE SC97: High Performance Networking and Computing, 1997
- Qiu, L., Zhang, Y. and Keshav, S., On Individual and Aggregate TCP Performance. 7th International Conference on Network Protocols, 1999.
- OProfile: <http://oprofile.sourceforge.net/news/>
- Linux TCP Tuning: <http://fasterdata.es.net/fasterdata/host-tuning/linux/>
- Sar manual page: <http://linux.die.net/man/1/sar>
- Iperf project: <http://sourceforge.net/projects/iperf/>
- Sysctl manual page: <http://linux.die.net/man/8/sysctl>
- FIO benchmark: <http://linux.softpedia.com/get/System/Filesystems/fio-7881.shtml>
- Bonnie benchmark: <http://www.textuality.com/bonnie/>
- Buffer bloat wiki: <http://www.bufferbloat.net/projects/bloat>
- Gettys, J., "Buffer bloat: dark buffers in the Internet", talk at Bell Labs. April 3, 2011.

A SLA-based Framework with Support for Meta-scheduling in Advance for Grids

Javier Conejero¹, Blanca Caminero¹, and Carmen Carrión¹

¹Albacete Research Institute of Informatics(I³A)

University of Castilla–La Mancha. Campus Universitario, 02071, Albacete. Spain

Abstract—*Quality of Service (QoS) is one of the most important and active research topics within Grid technology. But the emerging transformation from a product oriented economy to a service oriented economy establishes a new scenario where actual QoS mechanisms need to be enforced and new QoS mechanisms needed. Service Level Agreements (SLAs) are considered the cornerstone born to fulfill those requirements and the key concept that boosts the Grid economic exploitation. Therefore mechanisms to negotiate and manage SLAs become necessary.*

The aim of this paper is to address QoS within Grids by proposing an architecture capable of providing a service level agreement negotiation service and management for the agreed terms.

Our proposal introduces two new layers on the top of a Grid architecture with scheduling in advance feature, responsible of the SLAs handling and management. These layers consider the SLA standard proposed by the Open Grid Forum (OGF). In addition, they can also handle meta-scheduling mechanisms for non trivial execution parameters (e.g. economical, energetic, etc.).

Resulting as a potential improvement over QoS that SLAs in coordination with scheduling in advance can achieve within Grids. It is also shown the flexibility of this architecture and how it can be easily adapted to work over different Grid middlewares improving the interoperability of heterogeneous Grids.

Keywords: Service Level Agreement, Quality of Service, Grid Computing, WS-Agreement, Scheduling in advance.

1. Introduction

With the emerging interest on Cloud Computing and the new paradigm that it has introduced into Distributed Computing, QoS and *pay-per-use* models, the economy is transforming from a product oriented economy to a service oriented economy. This trend is boosting the economic exploitation of Distributed Computing environments like Grid Computing and High Performance Computing [1]. Therefore, new mechanisms are needed in order to evolve and adapt to the new needs.

Grids, Clusters and their combinations have been rediscovered with this new trend because of the business interest

on those technologies. Enterprises are interested on exploiting the resources they own in order to get benefits. This pushes research to go on this direction and Service Level Agreements (SLAs) and their management are intended to fulfill these needs [2].

The exploitation of these technologies is defined by enterprises and it can be economic, time scheduled or related to other terms. So an agreement is needed between the two parts involved: user and provider. Thus, the negotiation and enforcement to fulfill the terms defined within a SLA are directly related with the QoS the user expects to receive.

The SLA concept within Grid environments can be defined as a contract between a user and a Grid service provider in which participants expectations and obligations are explicitly defined [3].

So, it can be said that SLAs represent a contract between the user and the Grid Service Provider where the QoS expected to be received from the Grid service provider and the legal implications are explicitly exposed.

The scope of this paper is to improve the QoS within Grids by proposing an architecture capable of providing a framework to support Service Level Agreements, negotiation and management of the agreed terms included on them.

The structure of this article is as follows: general concepts about SLAs are described in Section 2. The main problems in SLAs the proposed architecture is expecting to solve are pointed out in Section 3. Section 4 presents the architecture objective of this paper in detail. It also contains three subsections, the first two for the new layers proposed: SLA-Manager (4.1) and SLA-Backend (4.1); and the last one describing the underlying technologies, background and middleware (4.3). Related work is presented in Section 5. Finally, Section 6 concludes the paper and describes the guidelines for future work.

2. SLAs general concepts

The SLAs state of the art is mainly addressed by the WS-Agreement specification proposed by the Open Grid Forum (OGF) [4]. Previous work on this field was done by WSLA [5] and SLAng [6] specifications, but they are nowadays unmaintained. Due to enterprises behind the OGF interested on SLAs, that participate on its specification, it is defined as the most important these days; and consequently the actual standard.

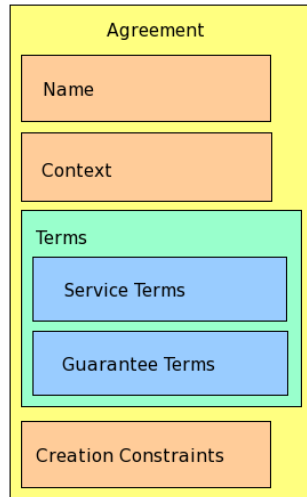


Fig. 1: SLA structure

WS-Agreement has been widely adopted by most projects where SLAs are on their roadmap [7], [8], [9], [10], [11], [12] but due to its limitations, alternatives have been developed causing the contrary effect of the definition of standard. Each project attempts to implant SLAs following the WS-Agreement specification adapting it for their private purposes. This causes, in some cases, incompatibility between them.

WS-Agreement is a Web Service protocol developed by GRAAP-WG from the OGF. It defines explicitly the structure of the SLA, a web service protocol for the SLA establishment between users and service providers, and the templates to make service provider discovery easier. Moreover, it defines the constraints, life-cycle, monitoring and runtime states for SLAs [4].

The structure defined by WS-Agreement for SLAs (Figure 1) [4] consists of four main blocks. The *Name* block only contains the name of the agreement which is optional and it is not the agreement identifier (just for human identification). The *Context* block contains all agreement details related to the context of the agreement (e.g. client id, provider id, etc.). These details are mandatory, but the specification lets the service provider extend them with new ones. The *Terms* block contains all terms related to assurances and commitments between user and service provider. The *Service Terms* are usually known as service descriptors, because they reference services or service terms properties (e.g. Number of CPUs, Amount of RAM, etc.). The *Guarantee Terms* specify the value or the QoS expected for the terms specified on the service terms (e.g. 4 (CPUs), 2 (RAM Gb), etc.). Finally, in the *Creation Constraints* block, the user can get some information from the service provider in order to know the limitations of the Service terms field. This block only appears on the negotiation template.

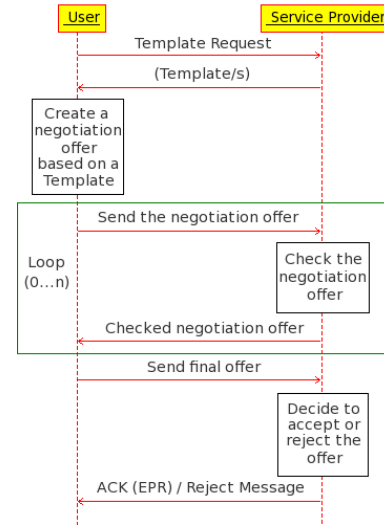


Fig. 2: WS-Agreement negotiation protocol with extension

Finally, the negotiation protocol defined by the WS-Agreement specification and updated by the WS-Agreement extension [13] (Figure 2) shows how users and service providers interact in order to negotiate an agreement defined by an SLA. It specifies a negotiation protocol with renegotiation support.

3. SLAs and QoS

The main issues to be tackled in the process of integration of SLAs on a distributed environment, in order to provide QoS guarantees, are mainly two: a) the extraction/choice of the QoS properties provided by the contract and b) to adjust the behaviour of the system according to them.

One of the main problems related to the first issue that appears when trying to integrate SLAs on a distributed environment are the temporal restrictions (e.g. start-time, deadline, etc. of jobs) that can be defined on an agreement. This problem is not tackled by the WS-Agreement specification and it is supposed to be addressed by other mechanisms.

Nowadays, some distributed systems have reservation in advance capability. This can solve the problem of temporal restrictions but due to the fact that reservations are not always possible, our proposal is going to be based on meta-scheduling in advance in Grid environments.

Meta-scheduling in advance can be defined as the first step of a reservation in advance process. In consequence, only job execution time periods and resources are selected, keeping track of them besides the resources status, but without making any physical reservation [14].

QoS in terms of temporal restrictions has been actively researched, mainly because metrics were known in advance (e.g. start-time) or could be evaluated from a profiling process (e.g. duration). And, as a consequence, job scheduling

based on these metrics can be highly optimized in reservation or scheduling in advance algorithms.

Furthermore, the integration of SLAs in Grid environments can be useful due to all the information contained on them. This information is obviated by most meta-schedulers (e.g. Condor [15], UNICORE [16], gLite [17], etc.) where only temporal metrics are used, preventing Grid environments from exploiting this information and, as a consequence, limiting the QoS reached.

But the use of other metrics, represented as terms on an agreement, could be used to take decisions that result in an improvement of QoS in terms of usage (e.g. number of CPUs needed, amount of RAM requested, etc.) or in terms of energy saving among others. Energy saving represents a highly active research topic within GreenIT [18]. In other words, these metrics could be used for a smarter job meta-scheduling system, managed by specific provider objectives.

This can not be managed by actual schedulers but it represents a very important source of information for the meta-scheduling process. So, the adjustment of the behaviour of the system according to them is needed in order to improve the QoS, as cited on the second issue.

Our proposal is presented to tackle these two problems by using both: a) a meta-scheduling in advance layer and b) a high level term meta-scheduling layer; such as will be detailed in next section.

4. Proposed Architecture

The following section shows the architecture proposed to solve the problems highlighted in Section 3.

Studying in detail the problems related with the adoption of SLAs in Grid Computing it can be thought that the SLA management, and by consequence all WS-Agreement specification, should be implemented into the middleware of the Grid. This could be a solution if the middleware has native support for reservation in advance or scheduling in advance as shown in Section 3. This is not the case in most Grid middlewares, so native support for SLAs and their management is not available on them. Therefore we propose to exploit the advantages given by the GridWay meta-scheduler enhanced with Scheduling in Advance Layer (SA-Layer) over Globus Toolkit 4 (GT4), which enables the scheduling in advance capability.

Our proposal consists of two new layers: SLA-Manager and SLA-Backend; solution which places on top of GT4 with the GridWay meta-scheduler and the SA-Layer (Figure 3) to handle SLAs on the Grid with scheduling in advance capability.

This solution has been designed to isolate the problems produced by the WS-Agreement specification and evolution (SLA-Manager), and it also isolates the problem of how to manage time limitations terms from others if we want them to participate on the meta-scheduling process (SLA-Backend).

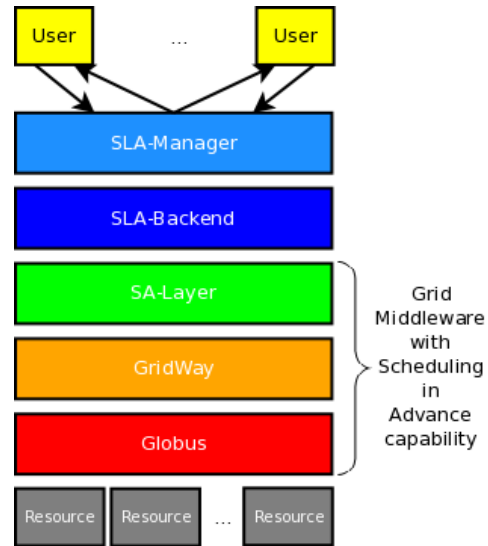


Fig. 3: Proposed Architecture

Furthermore, this solution has been designed for a not invasive implantation over the actual Grids and their structure if based on GT4. In other words, working Grids will not need to change anything from their actual configuration in order to extend their functionalities with WS-Agreement, just set the new layers in top of GT4. Service providers will be able to do a stagger transition of their services because with this proposal WS-Agreement will be working independently from its implantation.

To take advantage of the functionalities and libraries provided by the middleware, the SLA-Manager and SLA-Backend layers should be implemented as Web Services. This will make easier the security handling and, by defining well known interfaces, the use of this service from other web services (Figure 4).

Although the SA-Layer is external to GT4, the SLA-Backend must be able to interact with it and take advantage of it. GridWay is needed just to support SA-Layer requirements. The two layers proposed do not need interaction with

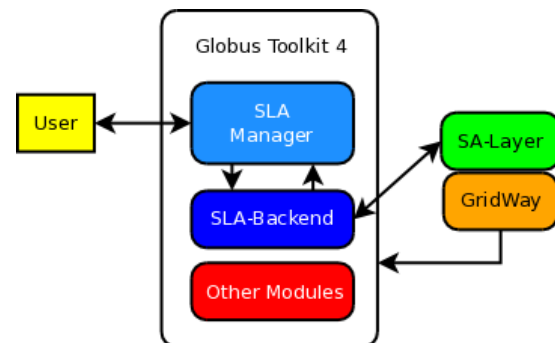


Fig. 4: Software design

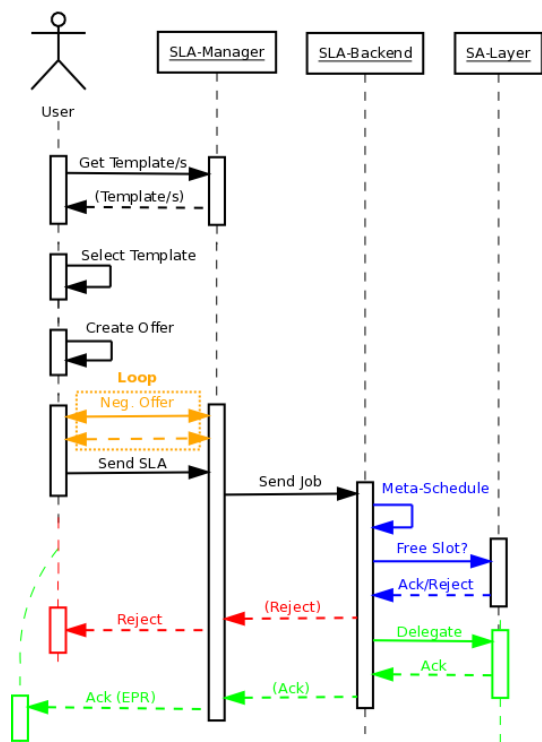


Fig. 5: Basic sequence diagram

it. If in some case, another meta-scheduler is desired to be used in place of GridWay and SA-Layer, the isolation is guaranteed and the only modification should be on the interfaces (if needed).

The workflow of our proposed architecture is shown in Figure 5. The user is who starts the negotiation process (as specified in WS-Agreement) by requesting a template. The user interacts with the SLA-Manager and performs the sequence diagram specified in WS-Agreement as shown in Figure 2. During this process the SLA-Manager interacts with the SLA-Backend in order to decide if the request done by the user is going to be accepted or rejected. If the request is rejected, the SLA-Backend layer communicates the decision to the SLA-Manager layer, which sends a reject message to the user. On the contrary, if the request is accepted, the SLA-Backend layer delegates the execution of the job on the SA-Layer and sends an acknowledgement message (ACK) with the corresponding endpoint reference (EPR).

On the next subsections, the building blocks of the proposed architecture will be described in detail.

4.1 SLA-Manager

The SLA-Manager layer has the mission of interacting with the user, offering the mechanisms to negotiate an agreement following the WS-Agreement specification.

It is designed to be implemented as a GT4 Web Service (Figure 4) so it can take advantage of the Web Services functionality (e.g. discovery, interfaces, etc.) and security. Security is essential when negotiating an agreement (e.g. trusted user?). So deploy the SLA-Manager as a GT4 Web Service can ensure security due to the Globus Security Infrastructure (GSI).

For this proposal, the SLA-Manager follows the WS-Agreement specification, but it can be easily extended for other purposes or extensions of the specification.

4.2 SLA-Backend

The SLA-Backend layer is defined as the intermediate layer between the SLA-Manager and the middleware infrastructure. Its main mission is to decide what to do with the requests performed by users.

These decisions are intended to be taken within this layer, so intelligence must be implemented in order to meta-schedule the incoming jobs.

This layer isolates the traditional scheduling decisions made by the middleware scheduler, or in our architecture, the SA-Layer meta-scheduler, which usually use time related parameters to take decisions.

The SLA-Backend sets a new higher level meta-scheduler that delegates time related parameters to be used on a lower level meta-scheduler (SA-Layer in our proposal), and it is thought to use any other (or more than one) metric/parameter like energy consumption, economy or any other term defined on a SLA.

So the SLA-Backend takes higher level decisions, letting the lower layers decide what to do with the jobs in terms of time-scheduling. This lets the service provider to have more control over the whole infrastructure in terms of human understandable metrics or parameters.

This module is also designed to be a GT4 Web Service to ensure security and to make the interconnection with the SLA-Manager easier (Figure 4).

4.3 SA-Layer and Middleware

Scheduling in advance has been chosen as the solution for the temporary restriction imposed by the use of SLAs in Grid environments. The reasons for this decision are mainly two: some kind of 'in advance' task is needed in order to solve the temporary restrictions (but scheduling in advance has been chosen instead of reservation in advance because reservations are not always possible); negotiation and renegotiation of SLAs can be extremely useful into scheduling and rescheduling in advance due to its not invasive reservation of resources. So, a middleware with this capability will represent the principal pillar in our proposal.

On a first approach, Globus Toolkit [19] middleware was chosen due to its "de facto" consideration. But its lack of scheduling in advance capability made us consider

another approach: use a solution that could give scheduling in advance capability to Globus Toolkit.

For our purpose, we are going to use the architecture presented on [14], [20] as base/middleware. On these works a layered architecture is proposed based on the use of Globus Toolkit 4, GridWay and the Scheduling in Advance layer (SA-Layer) innovation. This lets the system make the most of scheduling in advance capability.

The SA-Layer [14], [20] is the cornerstone of it. Using GridWay [21] functionalities like resource discovery and monitoring, work submitting and execution monitorization among others. The SA-Layer enables the scheduling in advance capability.

Moreover, the fact of being an architecture that respects the actual evolution of each module, isolating the tasks and responsibilities, makes this architecture the perfect base for our design.

On the contrary, this solution adds complexity to our approach, but as seen in this section, this does not mean a disadvantage.

This stack can be replaced by another one with scheduling in advance capability but it is necessary that it respects the interfaces and libraries defined on GT4 and SA-Layer interfaces for a swap without problems. Although wrappers could be designed in order to allow and facilitate this feature.

5. Related Work

Due to the importance of the topic exposed on this article, many efforts have been done on SLAs. For example, on their management [22], [23], QoS implications [24], semantic and virtualization exploitation [25] and specially on its standardization. The *Open Grid Forum* (OGF) proposes the use of WS-Agreement (*Web Services Agreement Specification*) [4], which is nowadays the most important and widely adopted.

Not only theoretical work has been done on this topic. Many projects have been interested on introducing SLA negotiation and management mechanisms on real Grids. Some of them will be analysed on the next paragraphs, focusing on their contribution to the topic.

AssessGrid (*Advanced Risk Assessment and Management for Trustable Grids*) [7] is a project that produced a free generic, configurable, trustable and interoperable software for the risk assessment, their management and decision support for Grid environments. It is freely available on the AssessGrid project web [7].

This project relies on SLAs. They are one of the most important aspects of this project, giving great importance to its management and negotiation. Its implementation is WS-Agreement based and developed for Globus Toolkit 4.

The AssessGrid project introduces a new innovation on SLAs field. The concept of a third entity into negotiation workflow: the broker. The user interaction with this entity is not mandatory, because if it were, they would not be able to talk about a WS-Agreement specification based

implementation. Moreover, it introduces new improvements to it: provider selection based on the best negotiation results, negotiation mediator (letting the broker distribute the incoming tasks specified on a SLA through the available service provider) and as a real negotiator (letting the broker negotiate instead of the user in order to get the best agreement) [26].

On this project, the broker service is in charge of solving the problem of job meta-scheduling and delegation. But as seen before, its mission is oriented to risk assessment.

It is important to mention that the negotiation protocol incorporated on this project respects the WS-Agreement specification, but it contains some other techniques that let them introduce the renegotiation concept and new negotiation workflows beyond the WS-Agreement negotiation extension [27].

Another interesting project at present is SLA4D-Grid (*Service Level Agreements for D-Grid*) [28], [9]. This project is designing and realising a SLA layer for the D-Grid (*Germany's National Grid*).

This project is based on the WS-Agreement specification. Furthermore, it is prepared to be easily extended for specific business purposes. It is implemented for Globus Toolkit 4 but designed for different middlewares (UNICORE [16], Globus [19] and gLite [17]) and services present on D-Grid.

Moreover, it has already produced a module known as Negotiation Manager which can offer WS-Agreement functionalities for Globus Toolkit 4 [29]. As expected, this project experienced the same problem exposed on this article. They solved it by using the ZIBARS module from AstroGrid-D project [30]. By this way they can handle scheduled jobs by delegating them to Reservation in Advance software (Maui meta-scheduler). Please note that this project does not support desktop Grids unlike our proposal.

WSAG4J (*WS-Agreement for Java*) [12] is a generic WS-Agreement specification framework developed by the Fraunhofer Institute (SCAI). It provides a quick and easy environment for the development of WS-Agreement based services. Moreover, it facilitates application and service debugging before their deployment.

It ensures that all application and services developed using this framework can be directly deployed into other environments if these respect the WS-Agreement specification.

Unlike AssessGrid and SLA4D-Grid projects, WSAG4J is implemented to work as an Apache Tomcat service. So, it does not let the user interact with a real Grid environment and it constitutes just a development framework.

Other projects like: Brein [8] and SmartLM [11] among others propose their own implementations of the WS-Agreement specification, but most of them modify it for their specific purposes that do not interfere with the basic operation. Thus possibly compromising the interoperability between Grids.

6. Conclusions and Future Work

This article presents a five layer architecture to empower the use of SLAs in Grid environments and solve the problems of job scheduling with advance timing requirements.

Moreover, it shows an architecture managed by a policy where SLA terms are on a higher level than execution parameters. So, specific schedulers can coexist in order to improve the QoS perceived by the user. In addition, new algorithms for these new high level term scheduler can now be designed and evaluated.

Furthermore, this proposal goes far away by not modifying the middleware and isolating the responsibilities in layers that can be easily adapted for private or business purposes.

This proposal pushes Grid environments to move on a more flexible management way where time restrictions not related parameters are involved in the job scheduling process. This help business exploitation of Grid resources by terms of economy and energy among others.

Some related work is analysed and results in a lack of a generic proposal that can handle SLAs where the scheduling in advance capability is available.

Future work steps are: implementation of the architecture proposed on this this article on a real Grid, and the design of tests to get empirical results regarding performance and usability.

Acknowledgement

This work was supported by the Spanish MEC and MICINN, as well as European Commission FEDER funds, under Grants "CSD2006-00046", "TIN2009-14475-C04" and through a FPI scholarship asociated to "TIN2009-14475-C04-03" project. It was also partly supported by JCCM under Grant "PII1C09-0101-9476".

References

- [1] D. Armstrong and K. Djemame, "Towards Quality of Service in the Cloud," in *Proc. of the 25th UK Performance Engineering Workshop*, Leeds, UK., 2009.
- [2] V. Stantchev and C. Schröpfer, "Negotiating and Enforcing QoS and SLAs in Grid and Cloud Computing," in *Proc. of the 4th International Conference on Advances in Grid and Pervasive Computing (GPC)*, Geneva, Switzerland, 2009.
- [3] J. Padgett, K. Djemame, and P. Dew, "Grid-Based SLA Management," in *Proc. of the European Grid Conference (EGC'2005)*, Amsterdam, The Netherlands, 2005.
- [4] A. A. et al, "Web Services Agreement Specification (WS-Agreement) GFD-R-P.107," Tech. Rep., March 2007, <https://forge.gridforum.org/projects/graap-wg/>.
- [5] "WSLA: Web Service Level Agreements," [Online]. Available: <http://www.research.ibm.com/wsla/>, Date of last access: 29th January, 2011.
- [6] D. D. Lamanna, J. Skene, and W. Emmerich, "SLAng: A Language for Defining Service Level Agreements," in *Proc. of the International Workshop of Future Trends of Distributed Computing Systems*, Los Alamitos, USA, 2003.
- [7] "AssessGrid," [Online]. Available: <http://www.assessgrid.eu>, Date of last access: 24th January, 2011).
- [8] EU-Brein - Bussiness Objective driven RELiable and Intelligen grids for real busiNess, [Online]. Available: <http://www.eu-brein.com/>, Date of last access: 24th January, 2011.
- [9] "D-Grid - SLA4D-Grid - Service Level Agreement für das D-Grid," [Online]. Available: <http://www.d-grid-ggmbh.de/index.php?id=89>, Date of last access: 27th January, 2011.
- [10] "SLA at SOI," [Online]. Available: <http://sla-at-soi.eu/>, Date of last access: 28th February, 2011.
- [11] "SmartLM," [Online]. Available: <http://www.smartlm.eu>, Date of last access: 25th January, 2011.
- [12] "WSAG4J - WS-Agreement framework for Java," [Online]. Available: <http://packcs-e0.scai.fraunhofer.de/wsag4j/>, Date of last access: 27th January, 2011.
- [13] O. W. et al, "WS-Agreement Negotiation Version 1.0," Tech. Rep., January 2011.
- [14] L. T. et al, "Network-aware meta-scheduling in advance with autonomous self-tuning system," *Future Generation Computer Systems*, vol. 27, no. 5, pp. 486 – 497, 2011.
- [15] "Condor Project: High Throughput Computing," [Online]. Available: <http://www.cs.wisc.edu/condor/>, Date of last access: 4th February, 2011.
- [16] "UNICORE: Distributed Computing and Data Resources," [Online]. Available: <http://www.research.ibm.com/wsla/>, Date of last access: 4th February, 2011.
- [17] "gLite: Lightweight Middleware for Grid Computing," [Online]. Available: <http://glite.cern.ch/>, Date of last access: 4th February, 2011.
- [18] G. Laszewski and L. Wang, "GreenIT Service Level Agreements," in *Grids and Service-Oriented Architectures for Service Level Agreements*, P. Wieder, R. Yahyapour, and W. Ziegler, Eds. Springer US, 2010, pp. 77–88. [Online]. Available: http://dx.doi.org/10.1007/978-1-4419-7320-7_8
- [19] I. T. Foster, *Globus Toolkit Version 4: Software for Service-Oriented Systems.*, ser. Lecture Notes in Computer Science, H. Jin, D. A. Reed, and W. Jiang, Eds. Springer, 2005, vol. 3779.
- [20] L. T. et al, "Meta-Scheduling in Advance using Red-Black Trees in Heterogeneous Grids," in *Proc. of the High Performance Grid Computing Workshop (HPGC), hold jointly with the International Parallel & Distributed Processing Symposium (IPDPS)*, Atlanta, USA, 2010.
- [21] C. V. et al, "Federation of teragrid, egee and osg infrastructures through a metascheduler," *Future Generation Computing Systems*, vol. 26, pp. 979–985, July 2010. [Online]. Available: <http://dx.doi.org/10.1016/j.future.2010.04.004>
- [22] J. P. et al, "Predictive Adaptation for Service Level Agreements on the Grid," *International Journal of Simulation Systems, Science and Technology*, vol. 7, no. 2, pp. 29–42, March 2006.
- [23] W. Theilmann and L. Baresi, "Multi-level SLAs for Harmonized Management in the Future Internet," *Towards the Future Internet*, pp. 193–202, 2009.
- [24] I. B. et al, "Advanced QoS Methods for Grid Workflows Based on Meta-Negotiations and SLA-Mappings," in *Proc. of the 3rd Workshop on Work ows in Support of Large-Scale Science. In conjunction with Supercomputing*, Austin, USA, 2008.
- [25] J. E. et al, "Exploiting semantics and virtualization for SLA-driven resource allocation in service providers," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 5, pp. 541–572, April 2010.
- [26] M. P. et al, "A Comparison of SLA Use in Six of the European Commissions FP6 Projects," Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, Tech. Rep. TR-0129, April 2008.
- [27] W. Ziegler, P. Wieder, and D. Battré, "Extending WS-Agreement for dynamic negotiation of Service Level Agreements," Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, Tech. Rep. TR-0172, August 2008.
- [28] "SLA4D-Grid Negotiation Manager," [Online]. Available: <http://www.sla4d-grid.de/>, Date of last access: 27th January, 2011.
- [29] M. Raack, "Documentation of the SLA4D-Grid Negotiation Manager (Globus)," Tech. Rep., July 2010, http://www.sla4d-grid.de/sites/default/files/SLA4D-Grid_Negotiation-Manager.pdf.
- [30] T. Röblitz, "Deliverable 5.6 -Resource Management for Grid-Jobs," Tech. Rep., May 2009.

CORS - A Cost Optimized Resource Reservation Scheme for Grid

Rifat Shahriyar, Md. Mostofa Akbar, M. Sohel Rahman, Md. Faizul Bari and Shampa Shahriyar

Department of Computer Science and Engineering (CSE)

Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh

Abstract—*The basic objective of grid computing is to support resource sharing among individuals and institutions within a networked infrastructure. Managing various resources in highly dynamic grid environments is a complex and challenging problem. Some approaches apply algorithms for resource management to grid but fails to provide any generalized solutions. Most of the approaches are based on a simple architecture considering computer as the main resource. But the real architecture of grid computing is a complex one especially if we consider various resources of a computer (e.g., processor, memory etc.) during resource management. In grids, sometimes assurance is needed for successful completion of jobs on shared resources. Such guarantees can only be provided by reserving resources in advance. So resource reservation is an integral part of resource management system for grid. Moreover the cost for providing resource as services will play a significant role in near future when resource sharing will be popular and inevitable. In this paper we provide a future reservation supported and cost optimized novel resource management system (CORS) for grid environment considering its real complex architecture. We further conduct a detailed performance evaluation with comparison on real workload traces for grid.*

Keywords: Grid Computing; Resource Reservation; Segment Tree; Parallel Workloads Archive

1. Introduction

Grid systems have emerged as promising next-generation computing platforms that enable the building of a wide range of collaborative problem-solving environments in industry, science and engineering [1]. The idea of grid computing was initially motivated by processing power and storage intensive applications. Its basic objective is to support resource sharing among individuals and institutions within a networked infrastructure. Resources that can be shared are processing capacity, storage, communication networks and bandwidth, data, software and licenses etc.

Managing various resources in highly dynamic grid environments is a complex and challenging problem. There exist works for resource management in different areas of computer science. Some approach uses data structures [2] [3] [4] and algorithms [5] for resource management to apply in grid but fails to provide any generalized solutions for grid environment. Most of the approaches are based on a simple

architecture considering computer as the main resource in their system. But the real architecture of grid computing is a complex one if various resources of any computer are to be taken into account during resource management. And indeed as the use of grid as a computing environment increases at a higher rate, these complex but real scenario must be taken into account. In grids sometimes assurance is needed for successful completion of jobs on shared resources. Such guarantees can only be provided by reserving resources in advance [6] [7]. So resource reservation is an integral part of resource management system for grid. Moreover grids are used as a voluntary service now a days. But with the recent improvements in architecture and usage, situation is predicated not be the same. Cost for providing resource as services will play a significant role in near future when resource sharing will be popular and inevitable. Clearly a complete resource management system for grid computing is required to support all the above mentioned features. This gives the motivation of this work where the goal is to provide a future reservation supported and cost optimized novel resource management system for grid environment considering its real complex architecture.

The main contribution of this work is a distributed, future reservation supported and cost optimized resource management system (CORS) for grid environment. Most of the existing approaches are based on a simple architecture considering computer as the main resource. But the real architecture of grid computing is a complex one especially if we consider various resources of a computer (e.g., processor, memory etc.) during resource management. That means various resources of a single computer can be shared by many participators in grid. Our system does that which is also one the contribution of this work. We perform a detailed performance evaluation of our prototype and compare it with an existing system using real workload traces. Superiority of our scheme is established from the comparative analysis presented on the experimental results on the workload traces. The rest of the paper is organized as follows. Section 2 illustrates our proposed resource management scheme and the data structures used by this scheme. Section 3 contains the experimental results along with comparative study against the state of the art system. We briefly conclude in Section 4 describing the key contributions of this work followed by some future research directions.

2. Proposed Resource Management Scheme of CORS

We proposed a new resource management scheme for grid computing environment considering the complex real life grid architecture. In this section, a detailed description of the system architecture of our resource management scheme is presented with an illustrative example. The proposed data structure is also described with example. We start with the problem statement in the next subsection.

2.1 Optimization Problem for the Resource Management Scheme

Grid applications can be broken down into a number of jobs. It is the responsibility of the job broker to break down the jobs of the applications. Each job of an application requires some grid resources to perform their operations. The participating computing nodes of grid usually provide the required resources of any job at a particular cost. So each of the resources of any computing nodes will have cost associated with it.

Let there be n applications in the grid termed as $A_1, A_2 \dots A_i \dots A_n$. An application A_i has a total of C_{A_i} jobs. Assume that the jobs of application A_i are $J_1, J_2 \dots J_j \dots J_{C_{A_i}}$. The participating computing nodes of the grids are $N_1, N_2 \dots N_k \dots N_l$ and the resources provided by them are $R_1, R_2 \dots R_r \dots R_m$. We assume that all the participating nodes will provide all the grid resources according to availability. To make the problem description simple, let us consider that the job J_j of the application A_i requires W amount of the resource R_r . The available amount of the resource R_r in the nodes $N_1, N_2 \dots N_l$ are $w_1, w_2 \dots w_l$ and the corresponding costs are $c_1, c_2 \dots c_l$. It is not always possible for a single computing node to completely serve a resource request. In most of the cases, a number of computing nodes jointly serve a resource request. The serving amount from the nodes are assumed as $s_1, s_2 \dots s_l$. If a particular node N_σ does not serve the job then the serving amount $s_\sigma = 0$. Now the total cost to serve a resource request is the sum of all individual computing nodes' service cost for their resource. So the total cost of the request will be $\sum_{k=1}^l c_k s_k$. The constraints need to be satisfied are as follows:

- 1) $\sum_{k=1}^l s_k = W$, i.e., a particular job gets exactly W amount of resource from the grid.
- 2) $\sum_{k=1}^l w_k > W$, i.e., there is available resource in the grid for a job.

Now the objective is to minimize the total cost $\sum_{k=1}^l c_k s_k$ to serve a resource request for a job of an application. Besides the objective of minimizing the cost it is also expected to reduce the number of participating nodes to deliver resource for a particular job. This will reduce the bottleneck for remote communication to the participating nodes. This additional objective can be formulated as follows:

minimize $\sum_{k=1}^l f(s_k)$ where

$$f(s_k) = \begin{cases} 1 & \text{if } s_k > 0 \\ 0 & \text{if } s_k = 0 \end{cases}$$

Here $f(s_k)$ is a boolean function indicating the presence of a node in serving a job.

2.2 Resource Management Scheme

The overall system architecture of our proposed resource management scheme (CORS) is shown in Figure 1. The components of the system, messages and their sequences to run the system are described by the caption of the blocks of Figure 1. Our proposed resource management scheme consists of the following phases:

Start Phase: The resource management will be controlled and coordinated by a set of computing nodes (computer) termed as Principal Resource Manager (*PRM*). The *PRMs* will be selected according to the grid administrators decision based on the configuration of the participating nodes. The *PRM* will be given a list of resources by the administrators that can be provided by the participating nodes of the grid environment. Each resource will be given a unique id named *ResourceId* for grid environment.

Initialization Phase: When a node wants to participate in the grid it will send a message named *msg_init* to any one of the *PRMs*. Each *PRM* has a list of participating computing nodes that will be synchronized amongst all the *PRM*. The *PRM* will accept the node and add it to the list. The node will be given a unique id named *NodeId* that will help to identify it in the grid environment. Each participating node will have a list of resources to provide service to the grid environment. This list will be a subset of the list maintained by the *PRM*. The given *NodeId* for any node is the same as the index of the node in the list maintained by *PRM*. Thus we can find the reference of any node through any of the *PRMs* in constant time. Each resource of a node will be given a *ResourceId* which is also the same as the index of the resource in the node's resource list. Thus we can find the reference of any resource of a given node in constant time. Each node can be considered as its own resource manager (*RM*). Any participating computing node can be selected as *PRM*.

Request Phase: Any application on the grid can be broken down into a number of jobs. An application sends a message named *msg_app* to job broker so that job broker can break it down into a number of different jobs. The jobs usually request resources from the grid. The request will be initiated by the job broker. Job broker will forward the request to any one of the *PRMs* so that the request processing is distributed among the *PRMs*. The request mainly contains resource identifier, starting time, and ending time. The *PRM* will propagate the request to the all the participating nodes. . Any single job can issue request for multiple resources. Then the job broker can forward request

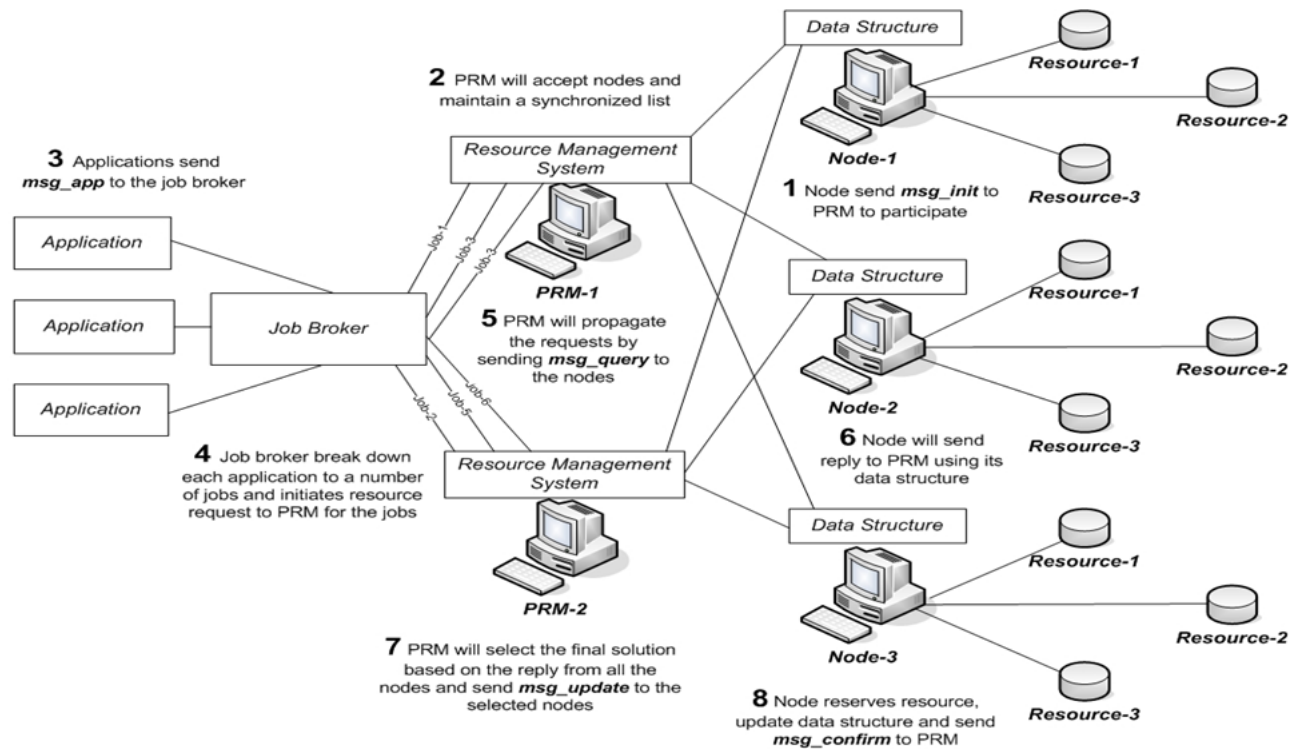


Fig. 1

ARCHITECTURE OF OUR PROPOSED SYSTEM CORS

for each resource to different *PRM*. So multiple *PRMs* can process request for a single job.

Search Phase: *PRM* will forward the request to each participating nodes by sending messages to the nodes. The messages sent to the nodes from *PRM* are generally named as *msg_query*. This is a parameterized message and based on the parameter a node replies with specific resource information, cost associated with a specific resource and available amount of specific resource in a given time frame. Some of the participating nodes may not provide the searched resource and they will be out of the search immediately. They will ignore the message. The nodes that provide the searched resource will receive the message. For each resource of each computing node, there will be an appropriate data structure to hold the information of used and available amount of resources in specific time frames. Then queries and corresponding updates will be carried out by the node in its own data structures. The data structure maintained by each participating node will not be replicated or copied to the *PRM*. The *PRM* will have only the reference of the nodes in the list and through that reference it can virtually have knowledge of the nodes' data structure. In this way multiple *PRMs* can have access to the most recent state of all of the resources without any space overhead. This is the main computation phase of our proposed resource

management scheme.

Reply Phase: After searching, the results will be returned to the *PRM* by the nodes. Each request will contain a *request_time* associated with it. *PRM* will wait for the result for a specified threshold amount of time from the *request_time*. Job broker will also wait for the replies from *PRMs* for a specified threshold amount of time from the *request_time* for the job that requires multiple resources. The result contains the notification whether the specific request can be served by this node or not. After getting the search result from all the nodes *PRM* will have a list of candidate nodes to serve the resource request. Now *PRM* needs to select the set of nodes that minimizes the total cost to serve the request. As we will show this problem can be mapped to well known fractional knapsack problem. The application of fractional knapsack problem in resource management is a novel idea for grid which we introduce here to guarantee cost minimization. Clearly the set of selected nodes will ultimately serve the request.

Reservation Phase: Once *PRM* has the list of selected nodes, it then sends a message named *msg_update* to each of the selected node to reserve required resource and update the data structure of the node. Upon receiving the message the node tries to update its data structure. If the update is successful then it will send a confirmation message named

msg_confirm to the *PRM* in reply. But sometimes updating may fail due to unavailability of resource as follows. The *PRMs* work in distributed manner. In the Search phase only available resource is searched but no update is made. So it may happen that another job acquires this specific resource of the node through any other *PRM*. That is why a confirmation ensures successful resource reservation.

For each resource of each computing node, there must be an appropriate data structure to hold the information of used and available amount of resources in specific time frames. Each element of the data structure will represent the (starting time, ending time, available amount) information for any resource. We know that the tree data structures are very much efficient for searching, inserting and deleting of elements. The segment tree structure, introduced by Bentley [8], is a balanced binary tree data structure that is used to store segments or intervals. We can map reservation supported resource management problem of our system to the segment tree with a little modification. The s and t , with $s < t$, of the segment tree $V(s, t)$ can be mapped into the starting time and ending time of a session where starting time < ending time. We add a field (available resource amount of a segment) to each leaf node. So each leaf node in the segment tree will contain starting time, ending time and amount of specific resource available (between the starting time and ending time). Leaves of the segment tree contain all the segments and the available resource amount. The internal node of the tree contains only the interval of its child node. The space usage of segment tree is $O(n \log n)$ where n is the total number of nodes and searching for a specific interval requires $O(\log n + k)$ time, where k is the number of reported segments. It does not depend on the number of intervals.

2.3 Fractional Knapsack Problem

In the fractional knapsack problem we are given a set I of n items having weights w_1, w_2, \dots, w_n and costs c_1, c_2, \dots, c_n respectively. We need to select items from I , with weight limit K , such that the resulting cost (value) is maximum. Most of the Knapsack variants are NP-Hard problems and the greedy solution to these problem leads to suboptimal or approximate solution but a greedy strategy does provide optimal solutions to the fractional knapsack problem [9]. We map the optimization problem for the resource management scheme to fractional knapsack problem. A resource of a node can be considered as an *item* and its associated cost can be considered as *value*. We need to sort the resources of the nodes by increasing cost per unit resources as we need to minimize the total cost.

2.4 An Illustrative Example

Consider a grid environment where two Principal Resource Managers (*PRM*) are working named PRM_1 and PRM_2 . The provided resource list is $R = \{R_1, R_2, R_3 \dots R_n\}$. The participating node list is $N =$

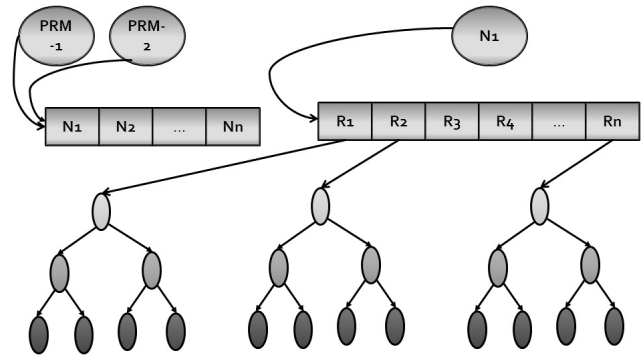


Fig. 2

OVERALL SYSTEM SCENARIO OF CORS

$\{N_1, N_2, N_3 \dots N_n\}$. Figure 2 depicts the overall scenario of the system. Here we can see the resource provided by a specific node N_1 . For each resource of N_1 there is a segment tree to maintain the available amount of resources in a specific time frame.

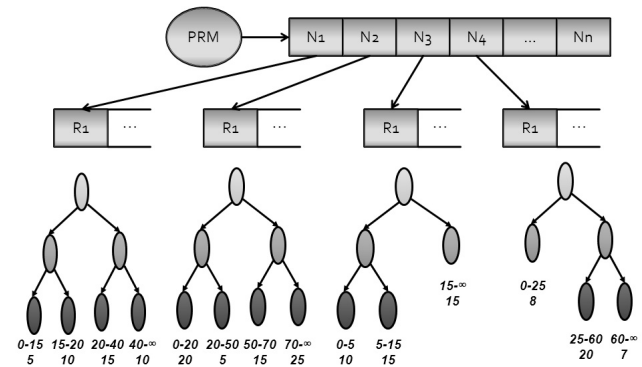


Fig. 3

NODE AND ITS DATA STRUCTURE

Let us consider that Application A_1 contains three jobs termed J_1, J_2 and J_3 . Now J_1 requires 25 units of resource R_1 for the time frame of (25, 45). At this point J_1 will send the request to the *PRMs*. Recall that *PRM* have a synchronized list of participating nodes. It will forward the query to the participating nodes. Figure 3 depicts the scenario of the nodes and corresponding resource R_1 . Here separate time intervals and corresponding available amount of resource is shown with the leaf nodes. As can be seen that Node N_1, N_2, N_3 and N_4 are providing resource R_1 . The corresponding data structure is also shown in the figure. Here the query will be $(R_1, 25, 45)$. Assume that the query is passed to each node data structure, the reply is listed in Table 1. After the search is completed, *PRM* will receive the above candidate list to serve the request for R_1 . Subsequently the nodes are selected according to the fractional knapsack solution, as

Table 1
CANDIDATE NODES AND AVAILABLE AMOUNT

Node	Amount	Details	Cost
N_1	10	minimum amount of time frame (20, 40) and (40, ∞)	4.0
N_2	5	amount of time frame (20, 50)	4.25
N_3	15	amount of time frame (15, ∞)	3.75
N_4	20	amount of time frame (25, 60)	3.5

Table 2
SELECTED NODES FROM THE CANDIDATE LIST

Node	Amount	Cost
N_4	20	$20 \times 3.5 = 70$
N_3	5	$5 \times 3.75 = 18.75$
Minimum Total Cost		88.75

shown in Table 2. So an add request will be sent to N_3 and N_4 and their corresponding segment tree will be updated as shown in Figure 4. The updated resource usages are shown in black shades. This concludes the resource reservation for the job. The application will then start running according to its starting time using these reserved resources. The details of the algorithms related to our proposed resource management scheme CORS and their complexity is not provided here due to page limitation. Their details can be found here [10].

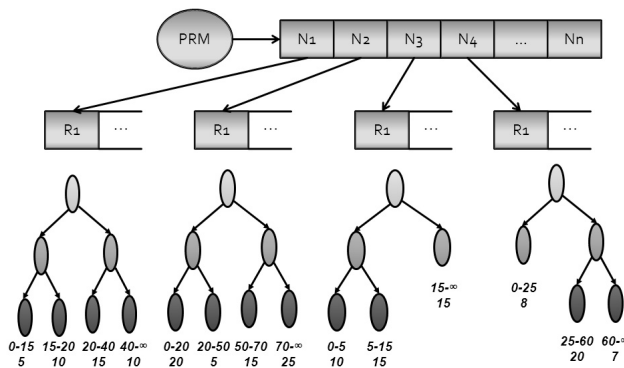


Fig. 4
NODE AND ITS DATA STRUCTURE

3. Experimental Results

In this section we present the experimental results of our proposed system. Through experiments we study the behaviour of our approach and evaluate its performance based on some performance metrics and also compare the performance of our approach to an existing system. A

simulator of the system is implemented using Java. The simulation is run using a computer having Intel Pentium-IV Dual Core 1.60GHz processor, 2 GB of memory and Windows XP operating system. We need to implement a new simulator because no existing simulator considers the complex grid architecture. In the existing simulators computer is considered as the only resource but in our system we need to consider various hardware and software of a computer as resources.

Node Selection Rules: The following rules are considered for assigning priority in selecting the next node to serve the request.

- **Max-Res:** This rule prioritizes the nodes that have maximum available resources. In this way number of connection establishment can be reduced.
- **Min-Res:** This rule prioritizes the nodes that have minimum available resources. In this way number of connection establishment can be increased.
- **Min-Cost:** This rule prioritizes the nodes that leads to the minimization of total cost.

Measurement Metrics: The metrics considered for evaluation are *TotalConnection* and *TotalCost*. We have also considered total memory consumption and running time.

TotalConnection: The term *TotalConnection* means the number of nodes required to completely serve a request. The requesting node needs to connect to these nodes. That is why we termed it as *TotalConnection*.

TotalCost: The term *TotalCost* means the total cost required to completely serve a request. This is the summation of all the individual cost of different nodes that serves the request.

Comparison with Sulistio's Resource Management Scheme on Real Workloads:

We compare our system CORS with an existing system for resource management in grid computing. The work done by Sulistio et al. [11] is the most appropriate to compare because it is the most recent work on resource reservation for grid. This work provides a new data structure for reservation using the Calendar Queue. There is no cost based framework exists for resource reservation in grid and this is also true for Sulistio's system. So we need to assume a default cost model. We developed Sulistio's system to minimize the *TotalConnection* by giving priority to the next device to be selected according to the rule Max-Res described before. It is guaranteed that Sulistio's system's *TotalConnection* will be minimized. On the other hand in our system we have been successful to achieve minimum cost solution to serve a grid request at the cost of a very small or no increase of *TotalConnection* from the minimum. Here we present results using real workload data for grid. Parallel Workloads Archive [12] contains an archive of information regarding the workloads on parallel machines and grids. It contains raw workload logs from various machines around the world. We choose three workloads, namely DAS2-fs0, LPC-EGEE and SDSC-BLUE. Details of the workloads are available at [12]. The main reason behind choosing

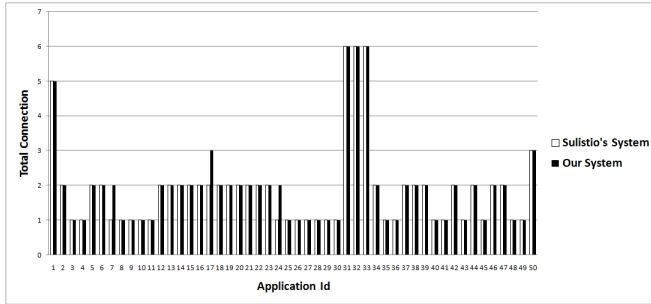


Fig. 5

TOTALCONNECTION REQUIRED FOR WORKLOAD DAS2fs0 USING SULISTIO'S SYSTEM AND OUR PROPOSED SYSTEM CORS

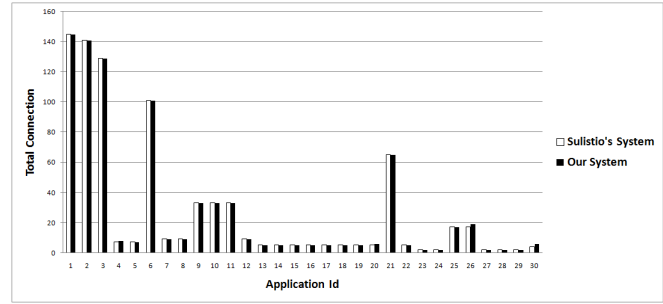


Fig. 7

TOTALCONNECTION REQUIRED FOR WORKLOAD SDSC-BLUE USING SULISTIO'S SYSTEM AND OUR PROPOSED SYSTEM CORS

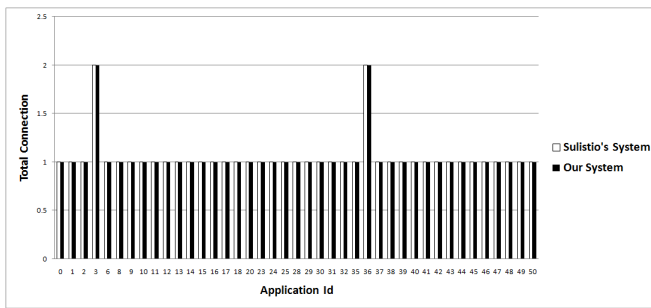


Fig. 6

TOTALCONNECTION REQUIRED FOR WORKLOAD LPC-EGEE USING SULISTIO'S SYSTEM AND OUR PROPOSED SYSTEM CORS

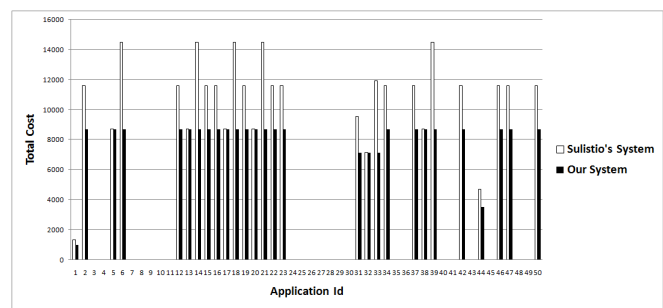


Fig. 8

TOTALCOST REQUIRED FOR WORKLOAD DAS2fs0 USING SULISTIO'S SYSTEM AND OUR PROPOSED SYSTEM CORS

these three is that these workloads have been considered by Sulistio in their simulation.

Evaluation with Respect to TotalCost and TotalConnection: We simulate our proposed system CORS and Sulistio's system using the above mentioned three workloads considering 50 sample applications.

In Figure 5 to Figure 7 we observe that the *TotalConnection* of Sulistio's system and our proposed system are equal for most of the applications. There are differences in *TotalConnection* for a few applications [3 applications in Figure 5 and 4 applications in Figure 7]. Here difference occurs for those applications whose jobs require huge amount of resources compared to the other applications of the same grid environment. We observe the presence of larger connection in Figure 7 compared to the other figures. It is also observed that in Figure 7 there are applications with various connection requirements. This justifies the high capacity of SDSC-BLUE and its multipurpose use by high, medium and low profile users in terms of resource requirement. In Figure 8 to Figure 10 we observe that the *TotalCost* of Sulistio's system is much greater than our proposed system as we expected.

Analysis of the Result: *TotalCost* of our system is guaranteed to be minimum as we use fractional knapsack to

minimize the total cost. But the interesting point is the margin of difference with the cost of Sulistio's system. The *TotalCost* of our proposed system is much less than Sulistio's system for all the workloads. *TotalConnection* of our system will not be minimum because we consider minimizing the *TotalCost*. But we tried to maintain *TotalConnection* as small as possible so that the increasing *TotalConnection* does not be a bottleneck. It is observed from the presented charts that we achieve the goal to maintain the difference as minimum as possible. For almost all the workloads *TotalConnection* for Sulistio's system and our proposed system are the same. This is because the nodes that provide the resources in a grid environment are mostly of same configurations and the jobs of the applications in a grid normally requires similar amount of resources. Grid applications are normally broken down into similar type of jobs by the job broker so that the application gets fair share of the resources. The details of how the job broker works is out of the scope of this research. However to justify the *TotalConnection* scenarios we briefly review it. In the grid environments most of the applications are similar in nature. So the job broker usually breaks down all the applications to same types of jobs where each job requires similar amount of resources. In such cases *TotalConnection*

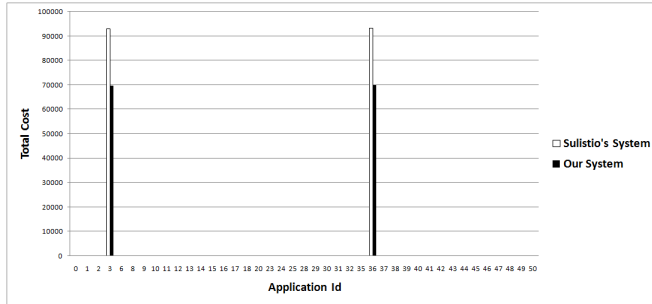


Fig. 9

TOTALCOST REQUIRED FOR WORKLOAD LPC-EGEE USING SULISTIO'S SYSTEM AND OUR PROPOSED SYSTEM CORS

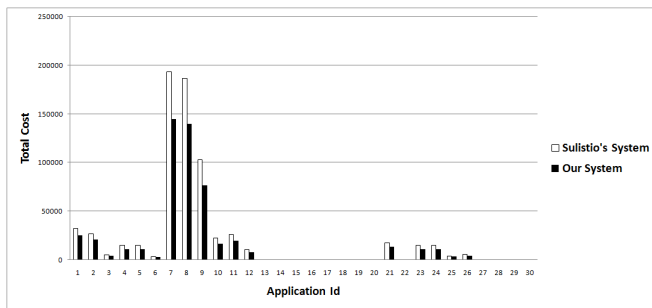


Fig. 10

TOTALCOST REQUIRED FOR WORKLOAD SDSC-BLUE USING SULISTIO'S SYSTEM AND OUR PROPOSED SYSTEM CORS

for Sulistio's system and our proposed system are the same. Sometimes there are exceptions in the workloads where a big sized application requiring huge amount of resource is broken down into a single job. This might happen due to the constraint that the application cannot be broken down into small jobs. Now consider a scenario where a particular node with the comparatively higher cost has the highest available resource. According to our proposed algorithm this particular node will not be chosen for consuming all the resources. But the Sulistio's algorithm will consume this resource to maintain *TotalConnection* minimum. That is why we observe substantial difference in *TotalConnection* in several exceptional cases. But generally it is observed that this difference is negligible. The memory consumption and running time of our proposed system CORS is also better than Sulistio's system. That means CORS memory consumption is lower and running time is less than that of Sulistio's system. The main reason behind this is the use of appropriate data structures and efficient algorithms in our proposed system. Due to page limitation the details of memory consumption and running time are not provided here. The details can be found here [10].

4. Conclusion

The main contribution of this work is a cost optimized complete resource management system with reservation support for grid computing. Resource management is not a new research area for grid computing but still there are lot of challenges and unsolved problems. Managing resources with negotiation is one of the open issues in grid resource management. Sometimes effective negotiation for flexible quality of service (QoS) can ensure more accepted jobs in grid system with full resource utilization. We have introduced a cost optimization model for resource management in grid computing. Future works can be done here to incorporate negotiation for cost between resource provider and resource requester (applications or jobs). The computing nodes that provide resource for grid also run local applications in their own operating environment. Works can be done how to optimally balance the distribution of resources for local and grid applications so that the local applications can not be affected by its services provided to the grid. Future works can also be done on resource management by considering the topology of the grid. In that case communication bandwidth requirement and latency will affect the resource management techniques.

References

- [1] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, CA, 1998.
- [2] L.-O. Burchard, "Analysis of data structures for admission control of advance reservation requests," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 3, pp. 413–424, 2005.
- [3] A. Brodnik and A. Nilsson, "Static data structure for discrete advance bandwidth reservations on the internet," *Computer Research Repository (CoRR)*, vol. cs.DS/0308041, 2003.
- [4] Q. Xiong, C. Wu, J. Xing, L. Wu, and H. Zhang, "A linked-list data structure for advance reservation admission control," in *In Proceedings of 3rd International Conference on Networking and Mobile Computing (ICCNMC)*, 2005, pp. 901–910.
- [5] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and A. Roy, "A distributed resource management architecture that supports advance reservations and co-allocation," in *In Proceedings of the International Workshop on Quality of Service*, 1999, pp. 27–36.
- [6] W. Smith, I. Foster, and V. Taylor, "Scheduling with advanced reservations," in *In Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2000*, 2000, pp. 127–132.
- [7] L. Yuan, C.-K. Tham, and A. L. Ananda, "A probing approach for effective distributed resource reservation," in *QoS-IP 2003: Proceedings of the Second International Workshop on Quality of Service in Multiservice IP Networks*. London, UK: Springer-Verlag, 2003, pp. 672–688.
- [8] J. Bentley, "Solution to klee's rectangle problems," *Technical Report, Carnegie-Mellon University, Pittsburgh*, 1975.
- [9] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.
- [10] R. Shahriyar, "http://teacher.buet.ac.bd/rifat/MSC.pdf," *A Distributed Optimized Resource Reservation Scheme for Grid Computing, M.Sc. Engg. Thesis, Bangladesh University of Engineering and Technology*, 2010.
- [11] A. Sulistio, U. Cibej, S. K. Prasad, and R. Buyya, "Garq: An efficient scheduling data structure for advance reservations of grid resources," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 24, no. 1, pp. 1–19, 2009.
- [12] P. W. Archive, "http://www.cs.huji.ac.il/labs/parallel/workload."

Dynamic and Decentralized Approaches for Optimal Allocation of Multiple Resources in Virtualized Data Centers

Wei Chen, Samuel Hargrove, Heh Miao, Liang Hong
(wchen, skhargrove, hmiao, lhong)@tnstate.edu

College of Engineering, Technology and Computer Sciences
Tennessee State University, 3500 John A. Merritt Blvd, Nashville, TN 37209, U.S.A.

Abstract: *In this paper, we propose dynamic and decentralized approaches for optimally allocating multiple resources in virtualized data center that has time-varying workload and heterogeneous applications. Instead of using predication based approaches or sensor measurement based approaches for resource provision, in this work, we tackle the problem with market based approaches that simplifies the control scheme and enable real-time control decision making based on each server's queue information. The proposed resource allocation scheme combines local optimization and heuristics for global optimization. In order to avoid high complexity related to multiple resources and multiple applications, we use a simple reinforcement learning method to achieve unknown optimal resource utility level. The simulation results show that our approaches can jointly maximize the throughput of the applications and minimize the usage of the resources. Furthermore, our approaches can adapt to unpredictable changes in the workload and do not require prediction or measurement of the utility level of different resources.*

Keywords: *cloud computing, resource allocation, optimization, distributed algorithms*

1 Introduction

Cloud computing features a shared computing infrastructure that hosts multiple applications. Since resource multiplexing leads to efficiency, the shared virtualized infrastructure is a paradigm for achieving the complex enterprise service applications that have time-varying demands on multiple resources. However, it is challenging to reduce the infrastructural and operational costs in the data centers while simultaneously increasing resource utilization to meet service requirements by taking into account that resources are dynamically shared and applications are unpredictable interacted across

A growing number of studies have been reported in the literature to improve the efficiency of the resource utilization in the large-scale cloud computing

environments. In [4], the resource allocation in cloud computing is formulated via a market model and solved as a conventional static scheduling problem. In this approach, each service is assigned by a set of resources and a number of contiguous timeslots. However, the resources cannot be used for other services during the processing period. Due to the shared structure and time-varying demands in cloud computing, more and more research works have focused on dynamic approaches. In [1] and [6], the bandwidth is sliced and assigned to virtual machines at each server according to the network topology and bandwidth demands. These approaches require a TCP transport mechanism and traffic predictions. On the other hand, the power management is also one of the critical issues in enterprise data centers. Recent studies indicate that the lifetime costs associated with the power consumption, cooling requirements, etc. of servers are significant [5]. As a result, there have been numerous works on power management in the data centers (see [8] and references therein). The traditional techniques use a closed-loop control model where the objective is to converge to a targeted performance level by taking control actions. It cannot be used for utility maximization problem in power management where the targeted optimal value is unknown. In [2], a greedy resource allocation algorithm is proposed that can adjust resource prices to balance the supply and demand, and allocate resource to their most efficient use. This

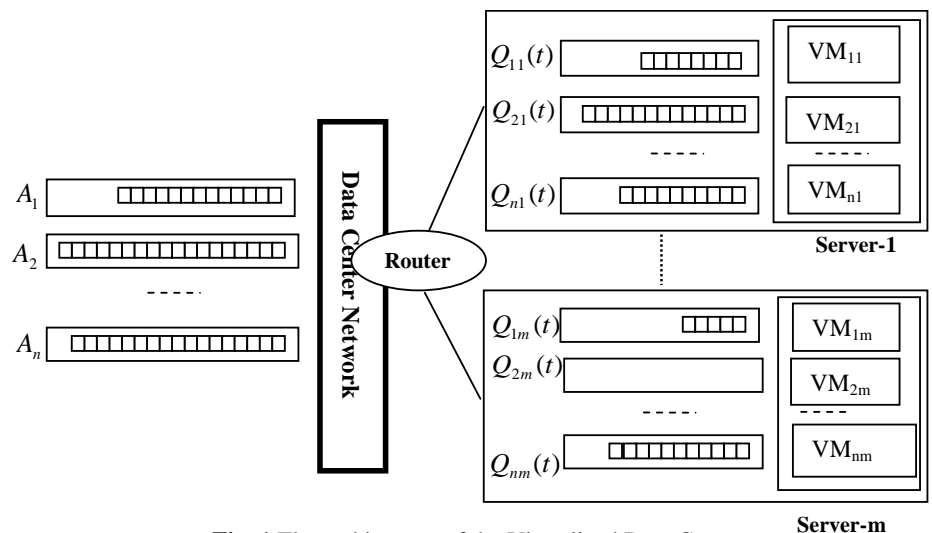


Fig. 1 The architecture of the Virtualized Data Center

approach depends on a pre-decided targeted utilization level which cannot be easily estimated. If the targeted utilization level is estimated too high, power will be wasted; otherwise, the performance will be degraded. In [9], a power management system including online admission, routing, and resource allocation is proposed. The power allocation is based on the queue information. However, this approach is limited to power allocation only. In [7], power and disk are co-allocated with the ability to automatically adapt the resources to dynamic workload. However, the system needs sensors that periodically collect statistics of real-time utilizations and performance of CPU and the disk. The behavior of sensors and actuators affect the efficiency of the resource utilization. As the result, the paper pointed out that many actuators are poorly designed.

In this paper, we deal with the problem of optimal multiple resource allocation in virtualized data center with time-vary workload and heterogeneous applications. Instead of using prediction based approaches or sensor measurement based approaches for resource provision, we formulate the problem by a market based approach that simplifies the control scheme and enables real-time control decision making based on queue information. The proposed resource allocation scheme combines local optimization and heuristics for global optimization. In order to avoid high complexity related to multiple resources and multiple applications, we use a simple reinforcement learning method to achieve unknown optimal resource utility level. The simulation results show that our approaches can jointly maximize the throughput of the applications and minimize the usage of the resources. Furthermore, our approaches can adapt to unpredictable changes in the workload and do not require prediction or measurement of the utility level of different resources.

2 Problem Formulation

2.1 Virtualized Data Center Model

We consider a virtualized data center with m servers that host n types of applications, where each server hosts a subset of the applications and provides a virtual machine (VM) for each application hosted on it [9] (Fig. 1). An application may have multiple instances running across different servers in the data center. We use an indicator $a_{ij} = 1$ or 0 to indicate application i is hosted or not hosted on server j for $i \in \{1, 2, \dots, n\}$ and $j \in \{1, 2, \dots, m\}$. Each server has a set of resources such as CPU, disk, memory, bandwidth, etc. Application requests arrive for each application i according to a random arrival process. We use A_i to indicate the queue of requests arriving for each application i . We assume a time-slotted system. At every timeslot, the requests in A_i are admitted into each buffer Q_{ij} at virtual machine VM_{ij} via network router. We assume that buffer Q_{ij} is large enough to keep the requests that will be processed in one timeslot.

2.2 Computing resources

Virtualized data centers enable consolidation of multiple applications sharing multiple resources. A customer can lease a virtual machine with a specific set of resources for a guaranteed service performance (e.g. the number of requests proceeded per second). The demands of applications are time-varying; therefore, more computing resources should be allocated on demand to an application when its workload incurs more resource demand. Among the computing resources at a server, for some resource such as power the total amount is adjustable, and for some resources such as disk, memory and bandwidth the total amount is fixed when the server is installed.

Power management is one of the critical issues in an enterprise data center. Modern CPUs can be operated at different speeds at runtime by using techniques such as Dynamic Frequency Scaling (DFS) and Dynamic Voltage Scaling (DVS). The power-frequency relationship is well-approximated by a quadratic model $P(f) = P_{\min} + \xi(f - f_{\min})$ [3]. In other words, based on time-varying demand, we can adjust the power consumption by operating CPU at different speed and runtime using DFS or DVS. When power demand is lower than the threshold, the server can be switched to sleep mode. For some resources, such as bandwidth, more resource does not always achieve the better performance. For example, assume that application i in virtual machine VM_{ij} at server j needs to transfer data to the applications p in virtual machines VM_{pq} at server q . If the outgoing bandwidth assigned to application i in VM_{ij} is much larger than the incoming bandwidth assigned to application p , the transfer rate will be very low since large amount of the transmitted data are lost at VM_{pq} and the lost data need to be retransmitted from VM_{ij} .

2.3 Market Model and Control Objective

In the data center, a customer leases a virtual machine with a specific set of resources for the guaranteed average throughput (number of requests per second). We define the profit at each timeslot t for the data center according to the payments of customers and the costs of resources. Assume that virtual machine VM_{ij} at server j hosts application i at timeslot t . Let R be the set of resources at the data center. At time slot t , let $\bar{r}_i(t)$ and $\bar{\lambda}_i(t)$ indicate the requested amount of resource $r \in R$ and average throughput (number of requests) that the user bids for, $r_i(t)$ and $\lambda_i(t)$ indicate the real allocated amount of resource $r \in R$ and real throughput, and $que_i(t)$ indicates the number of requests in queue Q_{ij} at virtual machine VM_{ij} . The profit in server j at timeslot t can be formulized as follows:

$$profit(t) = \sum_{i=1}^n (pay(\bar{\lambda}_i(t)) - \sum_{r \in R} r_i(t) \text{cost}(r)) \quad (1)$$

where the payment model is defined in the following formula:

$$pay(\bar{\lambda}_i(t)) = \begin{cases} bid(\bar{\lambda}_i(t)) - penalty_i(t) \\ \text{penalty case : if } \lambda_i(t) < \min(\bar{\lambda}_i(t), que_i(t)) \\ \& \exists r \in R, r_i(t) < \bar{r}_i(t) \\ bid(\bar{\lambda}_i(t)) + award_i(t) \\ \text{award case : if } \lambda_i(t) > \bar{\lambda}_i(t) \\ \& \frac{1}{T} \sum_{t'=t-T+1}^t \lambda_i(t') \leq \bar{\lambda}_i(t) \\ bid(\bar{\lambda}_i(t)) \quad \text{Others} \end{cases} \quad (2)$$

In formula (2), $bid(\bar{\lambda}_i(t))$ is the payment from the customer for application i and the guaranteed average throughput $\bar{\lambda}_i(t)$.

If $que_i(t) > \lambda_i(t)$, and $r_i(t) < \bar{r}_i(t)$ and $\lambda_i(t) < \bar{\lambda}_i(t)$, then the customer is under provisioned and the center needs to pay a penalty. If $\lambda_i(t) > \bar{\lambda}_i(t)$ and the real average throughput in time period T is smaller than $\bar{\lambda}_i(t)$ and the server successfully processed more requests than the requested average (the requests more than expected may caused by a sudden burst of the load), the center needs to be paid by an award. We suppose that the queues for application i admit the requests at the beginning of each timeslot. Let $que_i(t)$, $que'_i(t)$ indicate the number of the requests in the queue for application i at the beginning of timeslot t and at the end of timeslot t , respectively. By measuring the size of the queue, we can get real performance $\lambda_i(t) = que_i(t) - que'_i(t)$. The total profit from all servers j ($1 \leq j \leq m$) for all application i ($1 \leq i \leq n$) at timeslot t can be described as follows:

$$Tprofit(t) = \sum_{j=1}^m \sum_{i=1}^n (pay(\bar{\lambda}_{ij}(t)) - \sum_{r \in R} r_{ij}(t) \text{cost}(r)) \quad (3)$$

The control objective is to maximize the profit at each schedule slot by maximizing the payment from users and minimizing the cost of resources in the data center.

3 Decentralized Control Decisions

The control objective is to maximize $Tprofit(t)$. The time complexity for calculating the optimal solution is NP-hard, if all values of $r_{ij}(t)$ in formula (3) have to be considered. In this section, we first focus on the optimal resource allocation at one virtual machine, and then discuss the heuristics for the global optimization.

3.1 Localized Optimal Decisions

Consider application i that is hosted at server j with resource r . According to formula (1), the profit at timeslot t can be simplified as:

$$profit(t, i, r) = pay(\bar{\lambda}_i(t)) - r_i(t) \text{cost}(r) \quad (4)$$

To derive the penalty and award, we use $\rho_i(t) = \frac{bid(\bar{\lambda}_i(t))}{\bar{\lambda}_i(t)}$ to

denote the bid per request, and $\delta_i(t) = \frac{\lambda_i(t)}{\bar{\lambda}_i(t)}$ to denote the

degree of the real performance. The real performance is better than the required performance when $\delta_i(t) \leq 1$ or worse when $\delta_i(t) \geq 1$. Since the penalty depends on the difference of the required throughput and the real throughput and on the degree of the real performance worse than the required performance, we define $penalty_i(t) =$

$k(\bar{\lambda}_i(t) - \lambda_i(t))\rho_i(t)\delta_i(t)$, where $k \geq 1$ is a constant. On the other hand, for supporting a sudden burst of the load, the award is given when the real throughput is larger than the required average throughput but the real average throughput is not larger than that of the required average throughput. In order to avoid over using the resource, the award should not be over encouraged. We define $award_i(t) =$

$k'(\lambda_i(t) - \bar{\lambda}_i(t))\rho_i(t)$, where $k' > 0$ is a constant. $profit(t, i, r)$ can be derived from formula (4) and (2) to the following formula:

$$profit(t, i, r) = \begin{cases} bid(\bar{\lambda}_i(t)) - (k(\bar{\lambda}_i(t) - \lambda_i(t)) \frac{bid_i(\bar{\lambda}_i(t))}{\lambda_i(t)} - r_i(t) \text{cost}(r)) \\ \text{penalty case : if } \lambda_i(t) < \min(\bar{\lambda}_i(t), que_i(t)) \\ \& \exists r \in R, r_i(t) < \bar{r}_i(t) \\ bid(\bar{\lambda}_i(t)) + (k'(\lambda_i(t) - \bar{\lambda}_i(t)) \frac{bid_i(\bar{\lambda}_i(t))}{\bar{\lambda}_i(t)} - r_i(t) \text{cost}(r)) \\ \text{award case : if } \lambda_i(t) > \bar{\lambda}_i(t) \& \frac{1}{T} \sum_{t'=t-T+1}^t \lambda_i(t') \leq \bar{\lambda}_i(t) \\ bid(\bar{\lambda}_i(t)) - r_i(t) \text{cost}(r) \quad \text{stardard case : otherwise} \end{cases} \quad (5)$$

In (5), since $\bar{r}_i(t)$ is the amount of resource r that the user bids for average throughput $\bar{\lambda}_i(t)$ and the bid for a unit of the resource must be larger than the cost, we have $\frac{bid(\bar{\lambda}_i(t))}{\bar{r}_i(t)} \geq \text{cost}(r)$. When the performance is proportional

to the amount of the resource, we have $\frac{\lambda_i(t)}{\bar{\lambda}_i(t)} = c \frac{r_i(t)}{\bar{r}_i(t)}$, where

c is a positive constant. Since the constant k for the penalty is larger than 1 and the constant c for the ratio of performance

increase is close to 1, we have $\frac{k}{c} \geq 1$. In the formula of $profit(t, i, r)$, $\bar{\lambda}_i(t)$, $bid(\bar{\lambda}_i(t))$ and $cost(r)$ are constants given i , r and t . In formula (5), we replace $\frac{\lambda_i(t)}{\bar{\lambda}_i(t)}$ by $c \frac{r_i(t)}{\bar{r}_i(t)}$ in the penalty case and award case. The differential of $profit(t, i, r)$ on $r_i(t)$ can be derived as follows:

$$\frac{d(profit(t, i, r))}{d(r_i(t))} = \begin{cases} \frac{k}{c} bid(\bar{\lambda}_i(t)) \frac{\bar{r}_i(t)}{(r_i(t))^2} - cost(r) \\ \geq \frac{k}{c} bid(\bar{\lambda}_i(t)) \frac{\bar{r}_i(t)}{\bar{r}_i(t)r_i(t)} - cost(r) \frac{k}{c} \\ \geq cost(r) \left(\frac{\bar{r}_i(t)}{r_i(t)} - 1 \right) \geq 0 \\ \text{penalty case : if } \lambda_i(t) < \min(\bar{\lambda}_i(t), que_i(t)) \\ \& r_i(t) < \bar{r}_i(t) \quad \text{--- (6)} \\ ck' \frac{bid(\bar{\lambda}_i(t))}{\bar{r}_i(t)} - cost(r) \geq 0 \\ \text{award case : if } \lambda_i(t) > \bar{\lambda}_i(t) \\ \& \frac{1}{T} \sum_{t'=t-T+1}^t \lambda_i(t') \leq \bar{\lambda}_i(t) \\ - cost(r) \leq 0 \quad \text{standard case : otherwise} \end{cases}$$

Theorem 1 When the performance is proportional to the resource, i.e., $\frac{\lambda_i(t)}{\bar{\lambda}_i(t)} = c \frac{r_i(t)}{\bar{r}_i(t)}$, the function $profit(t, i, r)$ is monotone increasing on $r_i(t)$ in the penalty case and award case, and is a monotone decreasing on $r_i(t)$ in the standard case.

3.2 Heuristics for Global Optimization

In this section we describe the heuristics for global optimization. First, according to Theorem 1, when the performance is proportional to the amount of the resource, increasing resource r can raise $profit(t, i, r)$; otherwise, it may not. In order to avoid wasting the resource, when the condition does not hold, especially, increasing resource doesn't help to raise the performance, the amount of r should be reduced in next timeslot so that the reduced resource can be used for other applications in the same server, or even for other servers. Therefore, we have the following heuristic:

If $r_i(t) > r_i(t-1)$ & $\lambda_i(t) < \lambda_i(t-1)$, reduce resource r ----- (7)

As we described in Section 2.1, at the beginning of every timeslot, the buffers of virtual machines Q_{ij} in server j admit

the requests of application i from queue A_i . The servers can be heterogeneous and have different processing speed. For any two server p and q , buffer Q_{ip} has fewer requests left than buffer Q_{iq} indicates that server p processed more requests from application i than server q did. Therefore, Q_{ip} should admit more requests than Q_{iq} should in the next timeslot. On the other side, in order to save the resources the server in sleep mode should keep sleep if possible. It derives another heuristic:

At every timeslot, buffers $Q_{i1}, Q_{i2}, \dots, Q_{im}$ at the active servers j ($1 \leq j \leq m$) admit the requests from A_i fully in increasing order of buffers' sizes; buffers $Q_{i1}, Q_{i2}, \dots, Q_{im}$ at the inactive servers j admit the remaining requests in A_i if there is any. ----- (8)

4. Resource Allocation Algorithms

We use Theorem 1 to make the control decision on when to increase or decrease resources. Since the targeted optimal value for utility maximization is unknown, it is difficult to decide the amount of the resource for achieving the optimization. We use a form of reinforcement learning to adaptively achieve the unknown optimal utility level. For application i , the amount of resource r , $r_i(t)$, is decided as follows: $r_i(t) = r_i(t-1)\beta_i(r, t)$ ----- (9)

$\beta_i(r, t)$ is the estimated usage of resource r which is continuously adjusted in response to the needs of increasing or decreasing r as follows:

$$\beta_i(r, t) = \begin{cases} \alpha + (1 - \alpha)\beta_i(r, t-1) & \text{if resource } r \text{ is increased} \\ (1 - \alpha)\beta_i(r, t-1) & \text{if the resource } r \text{ is decreased} \\ \beta_i(r, t-1) & \text{Otherwise} \end{cases} \quad \text{----- (10)}$$

where $\alpha < 1$ represents the sensitivity of the WEMA filter. In a server, the total amount of resource r has a range and we suppose that $r_{\min} \leq r_i(t) \leq r_{\max}$. For the cases that the total amount of the resource does not change, $r_{\min} = r_{\max}$. For the resource such as power, we can set a threshold. If the power is lower than the threshold, the server is switched to sleep. For an application i , each resource r is allocated for i at each server by formula (8). If the total amount of resource r allocated to all applications at the server is larger than r_{\max} , then the amount of resource r for application i is adjusted to

$$r_i = \frac{r_i(t)}{\sum_{i=1}^n r_i(t)} r_{\max} \quad \text{----- (11)}$$

Now we consider the schedule for allocating multiple resources. In the data center, some applications are CPU contention; some are disk contention or bandwidth contention. The proportions of resources are unknown and difficult to predict. To solve the problem, we use round robin to allocate the multiple resources in R . We use $h=2|R|$ timeslots as one *scheduling* round, where each resource in R is allocated twice at two contiguous time slots. In other words, if we simply use r to represent the r th resource in R , in each scheduling round, resource r is allocated at $(2r-1)th$ and $(2r)th$ timeslots. For example, if power, bandwidth, disk are three resources in R , every scheduling round consists of 6 timeslots, and power is allocated at timeslots 1, 2, 7, 8, 13, 14, ..., bandwidth is allocated at timeslots 3, 4, 9, 10, 15, 16, ..., and disk is allocated at timeslots 5, 6, 11, 12, 17, 18, For allocating resource r at time slot t , the queue and performance information from two previous timeslots that allocated the same resource r are required. Therefore, in the above example, for allocating power at timeslot 8, the information from timeslot 2, and 7 are required. The outline of the resource allocation algorithm, called as Multi-Resource Allocation, that allocates resource r at timeslot t can be described as follows:

(1) **Admit arriving requests:** The arriving requests for each application i ($1 \leq i \leq n$) are admitted into buffer Q_{ij} of server j ($1 \leq j \leq m$) according to heuristic (8).

(2) Assuming that t' and t'' are the two previous contiguous timeslots (i.e., t'' , t' , and t are three contiguous timeslots for allocating the same resource r). At each server j , for each application i , the following step are executed:

(i) **Determine if resource r should be increased or decreased:** Use $\bar{r}_i(t')$, $\bar{\lambda}_i(t')$, $r_i(t')$, $\lambda_i(t')$ and $que_i(t')$ to judge the penalty case, award case and standard case according to the conditions given in formula (5). Resource r should be increased in penalty case or award case, and should be decreased in standard case. $r_i(t'')$, $r_i(t')$, $\lambda_i(t'')$, and $\lambda_i(t')$ are used to judge if the performance is proportional to the amount of the resource r ; if it is not, according to the heuristic (7), r should be reduced. Note that $t-1$ and t in (7) are t'' and t' here.

(ii) **Update the value of β :** Use formula (10) to update β according to whether the resource should be increased or decreased. Notice $t-1$ and t in (10) are t' and t here.

(iii) **Allocate the resource r :** In each server, use formulas (9) and (11) to calculate each resource r .

(iv) **Calculate the performance:** Calculate $\lambda_i(t)$, which is the difference of $que_i(t)$ at the beginning and end of the timeslot.

It is clear that the above algorithm for multiple resource allocation can be executed at each timeslot for each server in $O(n)$ time. Due to the space limitation, we omit the details of the algorithms.

5. Experiment and Evaluation

We evaluated our Multi-Resource Allocation Algorithm with three experiments through computer simulations. In the first experiment, power is the only resource to be considered and applications are homogeneous. Each CPU is assumed to have a discrete set of frequency options in the interval [1.6GHz, 2.6GHz] at increments of 0.2 GHz, where the minimum (maximum) power for operation is 120W (240W). We consider the scenario where 10 applications are hosted on 10 servers. On average, a server running at the minimum (maximum) speed can process 200 (400) requests/slot. If the power is lower than 120 W, the server is set to sleep mode. The average number of arriving requests for each application is 100 during 1 to 300 slots, 200 during 301 to 600 slots, and 300 during 601 to 900 slots. In the experiment, α is set to 0.1 and the initial value of β is 0.6. In Fig. 2, the dark blue (black if printed in black and white) and light pink (grey) lines are used to represent if the heuristic (8) is used or not in the algorithm. In both cases the throughput and power usage match the number of arriving requests in each of three durations. On average, in the first duration more than half servers are in sleep mode, in the second duration around one third servers are in sleep mode, and in the third period almost all server are in active mode. Fig.2 shows the throughput and total power consumption in the first experiment. We can see that the algorithm with heuristic (8) uses less power. Actually, it has a bit better throughput, too.

In the second experiment, bandwidth is the only resource to be considered. The experiment is executed by using NS2. There are 10 servers s1 to s10. Each has 0.375MB bandwidth. These servers transmit data to another server s that has 2.5MB bandwidth. In the experiment, each request transmits 1.25KB data. In the first scenario, we consider unbalanced workloads. There are average 100 arriving requests at each second for servers s1 and s2, 200 arriving requests at each second for servers from s3 to s8, and 300 arriving requests at each second for servers s9 and s10. In Fig. 3, the top strip shows the throughput of s9 and s10, the middle strip shows the throughput of s3 to s8, and the lowest strip shows the throughput of s1 and s2. It shows that the algorithm allocated the bandwidth exactly according to the demands. In the second scenario, we consider workload bursts. The arriving requests at each server are scheduled as Table 1. In server s3, the number of arriving requests is 300 during 60 to 70 seconds that is larger than the request performance (200); however it is 200 on average because the number of requests during 30 to 60 seconds is smaller than 200. Therefore, all the 240 requests should be performed during 70-80 seconds in s3 otherwise the real performance will be larger than the requested performance even in average. It is easy to see that in some periods, the total demand of bandwidth is larger than 2.5MB, the bandwidth at server s. Fig. 4 shows the throughput at server s3 (we don't have the space to show the figures for all servers).

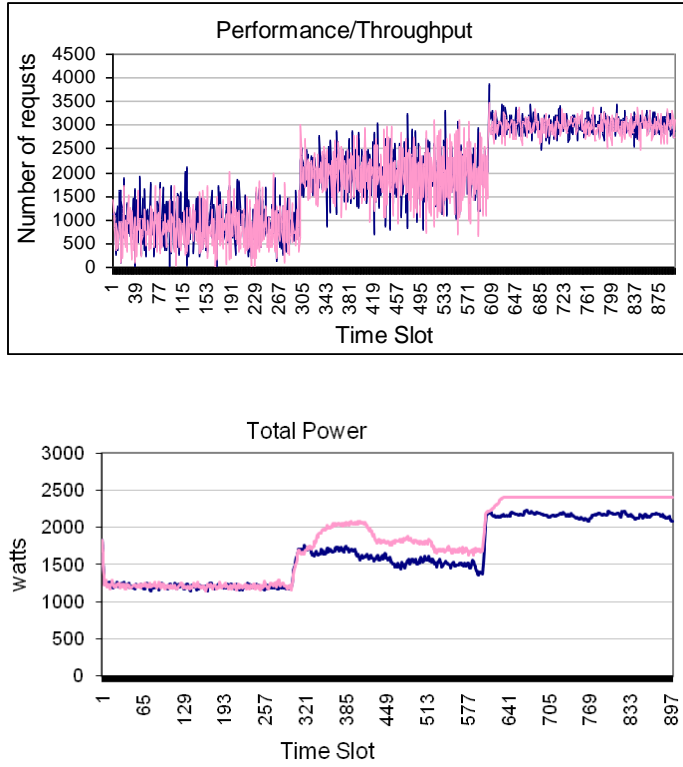


Fig. 2 Throughput and power consumption in the first Experiment

We can see that comparing with TCP, our algorithm allocate the bandwidth much more precisely and closer to the demands.

In the third experiment, power and bandwidth are both considered. In this experiment, we have 5 servers hosting 4 heterogeneous applications A1 through A4 with time-varying workloads. We adopt these applications from [7]. In A1, each request gets 10KB files from the database and does encryption, and it uses 0.095% of the full power (240W). In A2, each request gets/sends 80KB files from/to the database without doing encryption, and it uses 0.233% of the full power. A3 and A4 are Web applications. A3 is an auction-web tier. Each request uses 0.53 % of the full power and 10.3KB bandwidth. A4 is an auction-database tier. Each request uses 0.2% of the full power and 5.2KB bandwidth. Requested performance for each application is 400 requests per second. The bandwidths for the five application host servers, the web server and the database server are all set to 70MB. The arriving requests at the data center are scheduled as Table 2.

Table 1 Schedule of Arriving Requests at Each Server in Experiment 2

Time (seconds)	0-10	10-20	20-30	30-40	40-50	50-60	60-70	70-80	80-90	90-100	100-110	110-120
Arriving frames at Server 1&2 (Request Performance = 100)	50	50	200	200	100	100	50	100	100	100	150	400
Arriving frames at Server 3 to 8 (Request Performance = 200)	200	200	200	160	150	150	300	300	200	200	150	150
Arriving frames at Server 9&10 (Request Performance = 300)	300	300	200	400	300	300	50	180	250	250	400	450

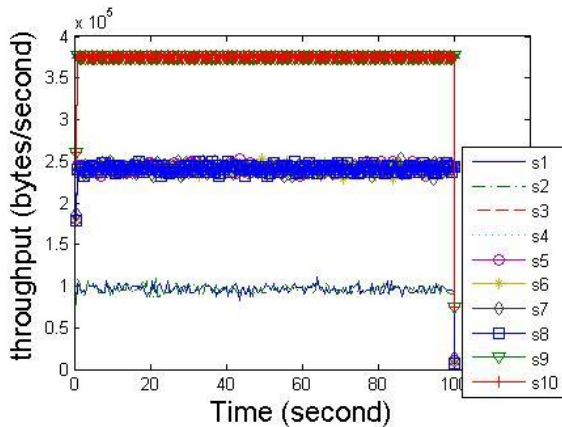


Fig. 3 Throughput of s1 to s10 for unbalanced workloads

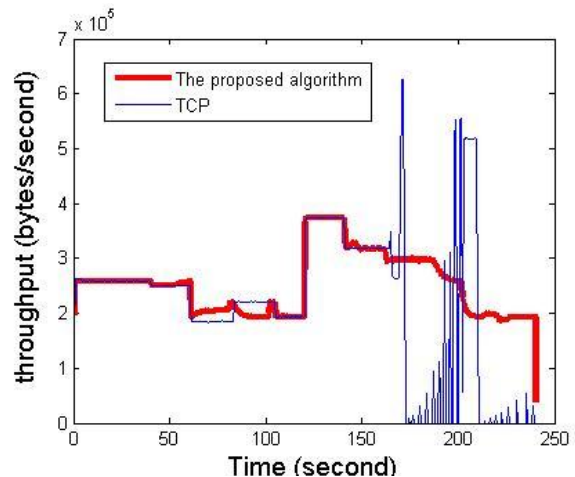


Fig. 4 Throughput of s3 for workload bursts

Table 2 Schedule of Arriving Requests for Each Application in Experiment 3

	1 – 50s	51 – 100 s	101 – 150 s	151 – 200s	201 – 250 s	251 – 300 s
A1	300	700	500	250	500	750
A2	300	500	700	500	300	750
A3	300	300	450	600	500	750
A4	300	500	300	500	700	750

The workload is unbalanced and bursts in the above table. Fig. 5 shows the performance of each application in all five servers that matches the numbers of arriving requests. For example, during 51-100 seconds, the number of arriving requests (700) for A1 is larger than the requested performance (400). However, since the average number of arriving requests is not larger than 400, all 700 requests are processed. In the last 50 second, the number of the arriving requests for each application is too large that it is beyond the power and bandwidth that each server can deal with. Therefore, only 675 requests on average for each application are processed instead of 750 arriving requests. On the other hand, during the first 50 seconds, only 300 requests arrive for each application. Therefore, two servers are at sleep mode during this period to save power.

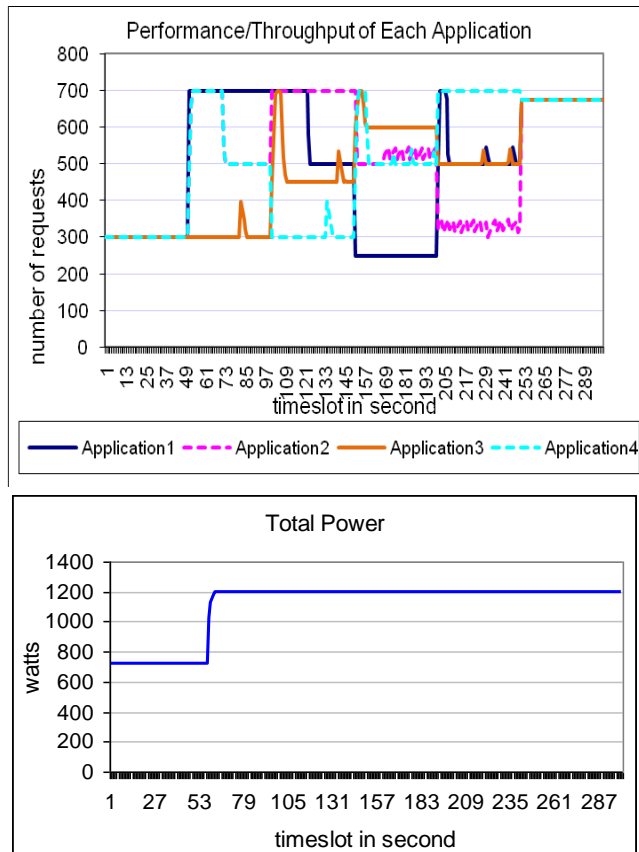


Fig. 5 Performance and power usage in the third Experiment

6. Conclusion

In this paper, we proposed dynamic and decentralized approaches for allocating multiple resources in virtualized

data center that has time-varying workload and heterogeneous applications. We tackled the problem with market based approaches that simplified the control scheme and enabled real-time control decision making. The proposed resource allocation scheme combines local optimization and heuristics for global optimization. The experiment results showed that our can effectively and efficiently allocate multiple resources in virtualized data center with time-varying workload and heterogeneous applications. They are adaptive to unpredictable changes in the workload and do not require prediction or measurement of the utility level of resources.

Acknowledgment

This research is partially supported by IBM faculty award in 2010.

References

1. O. Beaumont, H. Rejeb, "On the importance of bandwidth control mechanisms for scheduling on large scale heterogeneous platforms," Proceedings of IEEE International Symposium on Parallel and Distributed Processing, 2010.
2. J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, "Managing energy and server resources in hosting centers," Proceedings of SOSP, 2001.
3. X. Fan, W. Weber, L. Borroso, "Power provisioning for a warehouse-sized computer," Proceedings of ISCA, 2007.
4. I. Fujiwara, K. Aida, I. Ono, "Market-based resource allocation for distributed computing," Vol. No.34, IPSJ SIG Technical Report, 2009.
5. A. Greenberg, J. Hamilton, D. A. Maltz, P. Patel, "The cost of a cloud: research problems in data center networks," ACM SIGCOMM Computer Communication Review, vol. 30, no. 1, 2009.
6. T. Lam, G. Varghese, "NetShare: virtualizing bandwidth with the cloud," Technical Report, the Computer Science and Engineering Department at the University of California at San Diego, 2009.
7. P. Padala, K-Y. Hou, K. G. Shin, "Automated control of multiple virtualized resources," Proceedings of EuroSys, 2009.
8. R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, X. Zhu, "No "power" struggles: coordinated multi-level power management for the data center," Proceedings of ASPLOS, 2008.
9. R. Urgaonkar, U. C. Kozat, K. Igarashi, M. J. Neely, "Dynamic resource allocation and power management in virtualized data centers," Proceedings of IEEE Network Operations and Management Symposium, 2010.

The Analysis for Virtualization Performance in Cluster and Cloud Computing

Ying-Chuan Chen, Shuen-Tai Wang, Hsi-Ya Chang, Te-Ming Chen, Chin-Hung Li

Software Technology Division

National Center for High-Performance Computing, NCHC

Tainan, Taiwan

{ ycc0301, stwang, jerry, gavin, OscarLi }@nchc.narl.org.tw

Abstract - Virtualization Technology is an interesting research topic in current cloud computing and service. Using the Virtualization Technology in cloud or cluster computing can obtain a lot of benefits, such as ability to deploy any virtual platforms rapidly, easiness to manage all precious resources, and cost reduction. In order to discover optimal performance for virtual platforms, several well-known virtual machines are evaluated by standard benchmark tools, including HPC Challenge benchmark and NetPIPE program. In our paper, we will analyze significant experiment results that not only demonstrate the adequacy of virtual machines for High Performance Computing, but also present different performance characteristics for virtualization on cloud environment.

Keywords: virtualization technology; cloud computing; cluster computing; virtual platform; performance

1 Introduction

Virtual platform service has been a popular issue in cloud computing and cluster computing. The major concept of Virtualization Technology (VT) is that users could build many kinds of virtual operation system (OS) and Guest OS on a physical machine, such as Windows and Linux. Besides, the Guest OS can be built by various virtual machine tools and correlated resources of physical system were shared, hence the performance of virtual machine (VM) [24, 29, 30, 31] becomes the most critical factor for virtualization mechanism.

In figure 1 shows the principal architecture of virtualization. Physical hardware resources were divided as virtual resources of virtual platforms by VM's monitors, and those virtual resources were assigned to each VM by different requirements. Moreover, The VM's monitor provides each Guest OS a set of virtual platform interfaces that constitute VMs, acting as a bridge to connect between hardware devices and VMs. Instructions were delivered to hardware layer from virtual platform, which receives results from monitor of VMs. Each virtual platform will run independently, although physical resources were shared. By the way, VM's monitor has two kinds of model, Hypervisor [17, 33] and Virtual

Machine Monitor (VMM) [20]. The main distinction between Hypervisor and VMM is that the former monitor runs above hardware layer directly with better performance than VMM, such as Xen and VMware's ESX. On the other hand, Microsoft's Virtual PC and VMware's Workstation adopt VMM as monitor of virtual platforms.

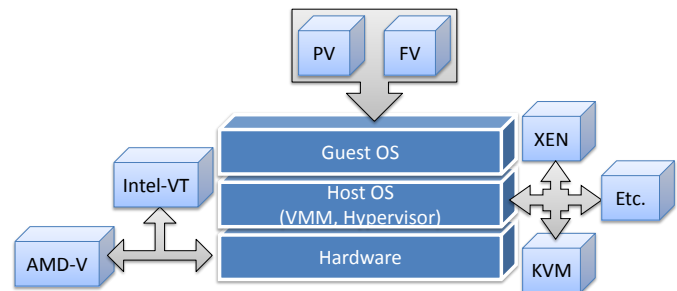


Figure 1. Virtualization Architecture.

For Guest OS, it includes two main virtualization [25, 26, 34] types: para-virtualization (PV) and full-virtualization (FV), which can be both combined with the hardware-assisted virtualization. The Guest OS is simulated by modified Kernel of Linux with PV, but related devices are not emulated. Instead, all devices are accessed through light-weight virtual drivers offering better performance and close to the physical machine. But the drawback is that guest kernels must be upgraded to provide new virtual system calls for the new services and all of Guest OS must be compatible with the Host OS.

Furthermore, it allows the execution of unmodified virtual OS by emulating physical system's hardware resources with the full-virtualization model. Many VM's tools adopted FV to build virtual OS. The advantage of FV is that it can build various virtual platforms in a physical machine. Nevertheless, FV may incur a performance penalty. The VMM of FV have to offer the VM through an image of entire system, including virtual memory space and devices, that also must create and maintain data structures for virtual components, such as a shadow memory page table, and these

data structures should be updated for every corresponding access with the VMs.

On the other hand, FV has an imperative cost which can be mitigated by using the hardware-assisted virtualization in the CPU device for providing the Guest OS with a complete physical system interface, consequently the hardware-assisted virtualization enables efficient FV using help from hardware capabilities, primarily from the host processors.

Both the Intel and AMD Corporations have supported hardware-assisted virtualization modules for their novel products of CPU. The Intel module, like the Intel-Virtualization Technology (Intel-VT) [6, 21, 32], combined with software-based virtualization solutions provides maximum system utilization by consolidating multiple environments into a single real machine. For AMD module, like AMD Virtualization (AMD-V) [1, 19, 35] Technology, allows users to better utilize related resources, which make virtual platforms more effective.

In contrast, PV provides better efficiency than FV for each VM through the virtualization abstraction of hardware that is similar but not identical to the underlying physical hardware. As a result, the Guest OS using PV technology is aware that allowing for near-native performance.

In fact, the performance of VM is certainly lower than physical machine. However, taking into consideration of a lot of advantages for deployment, management and cost, VM is the best solution for virtual service platforms in cloud computing area. Thus in order to obtain best performance with VM tools, we will analyze related VMs by benchmarks to present significant experiment results in our paper.

2 Virtual Machine

VT is able to apply not only to subsystems but to a complete virtual system. To implement a virtual machine (VM), developers design a software layer to real machines to support the desired architecture. By providing one or more efficient of entire virtual platforms, VMs have extended multi-processing systems of the past decade to be multi-environment systems as well. There are many kinds of VM in the market, but not all VMs fit to build virtual platforms, so we must choose a suitable VM to achieve our purpose. Common VMs include Xen, KVM, VMware and Microsoft Virtual PC; this section focus on these VM tools to describe the architecture in order.

2.1 Xen

Xen [15, 18, 27] is a famous open source virtualization software based on x86 hardware architecture by University of Cambridge. It provides both PV and FV mode to simulate one

or more complete virtual OS on single physical machine, and all of VMs must run on modified kernel with Xen.

Figure 2 illustrates the architecture of Xen, there is a hypervisor running on the hardware directly and acting as the interface for all hardware requests such as CPU, I/O, and disk for the Guest OS. By separating virtual resources from the physical hardware, the hypervisor is able to run multiple virtual OS securely and independently. Domain U means all of Guest OS which have deployed the above hypervisor layer, it includes Domain0, Domain1 and Domain2. Domain0 has many important instructs of control for Domain1 and Domain2, consequently it was built and loaded on physical machine firstly. The software of Xen has been adopted directly in newer versions of Linux, the user can install Xen's software fast and simply, such as Fedora 8, CentOS 5, etc.

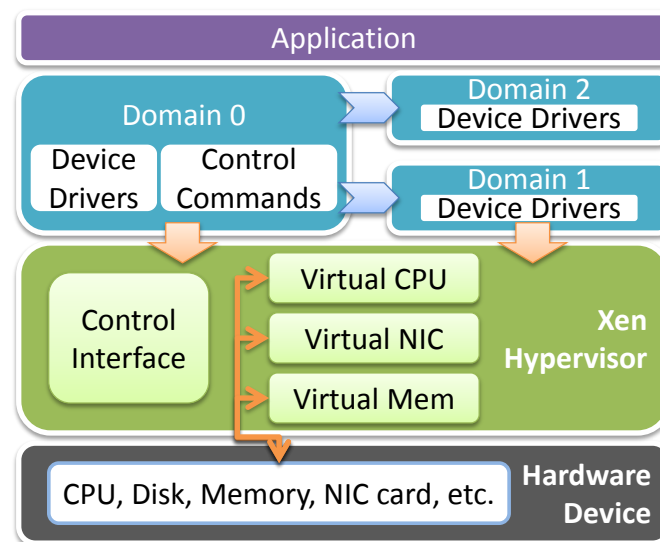


Figure 2. Architecture of Xen.

2.2 KVM

Kernel-based Virtual Machine (KVM) [7, 16, 22] is an open source software with GPL, developing by Qumranet company. KVM provide FV solution for Linux on x86 hardware containing virtualization extensions with Intel-VT or AMD-V, and the kernel component of KVM is encased in mainline Linux OS over version 2.6.20, hence the user can setup the virtualization environment of KVM as well as Xen when installing Linux OS conveniently.

KVM consists of a loadable kernel module called `kvm.ko`, that provides the core virtualization infrastructure and a processor specific module for `kvm-amd.ko` or `kvm-intel.ko`. KVM also requires a modified QEMU although work is underway to get the required changes upstream.

Using KVM, one can run multiple VMs running unmodified Linux or Windows images. Each VM has private

virtualized hardware devices, such as network card, disk, graphics adapter, etc. Besides, VM was like a process in queuing system of Linux, and then manager can directly kill any VMs by control command.

In order to easily manage all of VMs, KVM offered a visualization manage interface that is called virt-manager, and showing significant status of VMs like CPU, memory, disk, network, etc. Therefore the manager can add, delete, start, pause, stop, and destroy any virtual platform by the virt-manager tool.

2.3 Vmware

VMware Company [14] offers many powerful VM solutions, including VMware Workstation, VMware ESX, VMware ESXi, VMware Player and VMware Server; the latter three are free software.

The VMware virtualization platform uses software such as VMware ESX to transform the x86-based hardware resources, including the CPU, disk, memory and network, to build a fully functional VM that can run different virtual platforms similar to a real machine. Each VM contains a complete system, eliminating potential conflicts.

VMware virtualization works by inserting a thin layer of software directly on the hardware layer or on a host OS. This contains a VMM that allocates hardware resources dynamically. Multiple virtual OS run concurrently on a single physical machine and share physical resources with each other. Users can run various virtual OS safely at the same time, with each having access to the resources it needs when it needs them.

Nevertheless, the drawback is the high price to deploy the powerful VMware's solutions.

2.4 Microsoft Virtual PC

Virtual PC [8] is a virtualization software by well-known Microsoft Company, provides users the capability to run multiple virtual environments such as Linux or Windows OS from a Windows platform. Microsoft released the Windows-hosted version as a free product since July 2006, thus users can download free to install virtual OS on newer Windows 7 momentarily.

Virtual PC can simulate a standard x86 framework PC and its associated hardware devices on a physical machine. Unfortunately, it is only supported on host OS with Windows, and we can't set up virtual environments successfully base on Linux platform.

By comparing related capability between above different virtualization tools, there are two appropriate VM's software, including Xen and KVM, suited to build virtual platforms in consideration of cost, management, deployment, etc. Therefore, we will focus on Xen and KVM to analyze performance of various virtual platforms by some famous benchmark programs in our paper.

3 Benchmark

In this section, the paper will explain testing capability for related popular benchmark tools, including HPC Challenge benchmark and NetPIPE program.

3.1 HPC Challenge benchmark

The HPC Challenge (HPCC) benchmark [4] is maintained and managed by the Innovative Computing Laboratory at the University of Tennessee. It is a well-known suite of tests that includes the High-Performance Linpack, DGEMM, STREAM, PTRANS, RandomAccess, FFT and b_eff program has been adopted to evaluate various high-performance computing clusters as well as multiprocessor (SMP) architectures computers in the international website of TOP500 [13]. The HPCC benchmark consists of basically tests as follows:

- High-Performance Linpack (HPL) [5, 23] is a software package freely written in C that solves a linear system of equations in double precision arithmetic on distributed-memory computers. It provides three testing and timing modes, including Linpack100, Linpack1000 and HPL, to quantify the accuracy of the obtained solution as well as the time it took to compute it. But the evaluation ability of Linpack100 and Linpack1000 can't satisfy for calculating novel high-performance computers. Accordingly, the HPL mode was selected in recent years. Additionally, HPL requires the execution of the Message Passing Interface (MPI) and a math library, either the Basic Linear Algebra Subprograms (BLAS) or the Vector Signal Image Processing Library (VSIBL) is needed. The best performance achievable on a measured system depends on a large variety of factors, and users have to setup associated parameters in HPL.dat with conditions of evaluated distributed systems.
- DGEMM can used to measure the floating point rate (in Gflops) of execution of double precision real matrix-matrix multiplication.
- STREAM [12] is a synthetic benchmark program that measures sustainable memory bandwidth (in GB/s) and the corresponding computation rate for simple vector kernel.
- PTRANS (parallel matrix transpose) [10] is a useful test of the total communications capacity of the network. It can exercise the communications (in GB/s)

where pairs of processors communicate with each other simultaneously.

- RandomAccess [11] can measure the rate of integer random updates of memory. It is evaluated in GUPS (Giga Updates per Second) that is a measurement that profiles the memory architecture of a system and is a measure of performance similar to MFLOPS.
- FFT program [3] measures the floating point rate (in Gflops) of execution of double precision complex one-dimensional Discrete Fourier Transform (DFT).
- Communication bandwidth and latency based on the b_eff program (effective bandwidth benchmark) [2] is a set of tests to evaluate bandwidth (in GB/s) and latency (in usec) of a number of simultaneous communication patterns.

3.2 NetPIPE

The Network Protocol Independent Performance Evaluator (NetPIPE) [9, 28] is a free protocol independent performance tool that visually represents the network performance under a variety of conditions. It was programmed in C at the Scalable Computing Laboratory (SCL).

The NetPIPE performs simple ping-pong tests, bouncing messages of increasing size between two processes, whether across a network or within an SMP system. Message sizes are chosen at regular intervals, and with slight perturbations, to present a complete testing result of the communication system, such as throughput in Mega bits per second (Mbps) and latency in seconds. Each data point involves many ping-pong tests to show an accurate timing, and latencies are evaluated by dividing the round trip time in half for small messages.

4 Experiment

In this section, we perform HPCC benchmark and NetPIPE program practically to measure related performance of various virtualization platforms, including physical platform, Xen kernel, Xen PV, Xen FV and KVM FV, and important results of experiment will be shown as well.

A physical platform was built as the real cluster system with CentOS 5.5 OS. Xen kernel means the cluster system that was deployed with modified kernel (Domain 0) through Xen software, and Xen PV is the virtual environment for cluster that was organized via para-virtualization technology of Xen. The case using full-virtualization technology of Xen to construct the complete virtual cluster system is indicated as Xen FV. Furthermore, the cluster platform was simulated based on full-virtualization mode using KVM software is expressed as KVM FV in our paper.

The specification information of testing platform is showed in Table I. We adopt four HP servers as the hardware testbed, with two Quad-Core AMD Opteron(tm) Processor

2356 CPU in each server likes. The SATA 500GB disk of Western Digital (WD) is used to allocated 2GB size as swap space. There are 8GB memory and 1Gb Ethernet network interface card (NIC) in each measured environment, and the VM software versions for Xen is 3.0 and for KVM is kvm-83.

Table I. Specifications of Testbed.

CPU	Quad-Core AMD Opteron(tm) Processor 2356
Disk	Western Digital(WD) 500G SATA 7200RPM
Memory	8GB
Network	1Gb Ethernet
OS	CentOS 5.5 (2.6.18-194)
Swap	2GB
Xen	xen-3.0
KVM	kvm-83
Compiler	GNU
MPI	Mpich2

Table II illustrate the testing performance by using HPL for each evaluated platform, including Physical, Xen kernel, Xen PV, Xen FV and KVM FV.

Table II. HPL Performance of Platforms.

Platform Type	Rpeak (Gflops)	Rmax (Gflops)	Efficiency %
Physical		173.3	58.86
Xen kernel	294.4	172.3	58.52
Xen PV		161.1	54.72
Xen FV		49.65	16.86
KVM FV		93.89	31.89

Rpeak is the theoretical peak performance in Gflops, by counting the number of floating-point additions and multiplications that can be completed during a period of time. Due to CPU clock rate of each core at 2.3 GHz can complete 4 floating point operations per cycle, and there are eight cores in four servers respectively, thus Rpeak is 294.4GFlops that was calculated via formula (1):

$$\left\{ \begin{array}{l} R_{peak} = C \times F \times N \\ C = \text{CPU Clock Rate (GHz)} \\ F = \text{CPU floating point issue rate} \\ N = [\text{The number of processor}] \times [\text{The number of cores per processor}] \end{array} \right\}$$

Rmax is the maximum computing performance vaule of a platform in Gflops achieved in typical HPL program, and the notion of Efficiency is defined as the ratio between Rpeak and actual Rmax of the platform, that can be calculated through the formula (2).

$$Efficiency = \frac{R_{max}}{R_{peak}} \times 100\% \quad (2)$$

In table II, the Rmax value of Physical cluster platform is 173.3Gflops, and the rate of Efficiency is 58.86%. The Rmax values of Xen kernel and Xen PV are 172.3Gflops and 161.1Gflops, with Efficiency rate at 58.52% and 54.72% respectively, both decrease from the physical cluster's HPL efficiency slightly. For performance of Xen and KVM with FV mode, both Rmax and Efficiency values are much worse than previous cases. And the computing performance of Xen FV is lower than KVM FV since the ability of communication for network is the worst.

In Figure 3, all testing results of the NetPIPE program for every environment are obtained via the best throughput from ten rounds. Furthermore, the network throughput value in Mbps measured by Ping-pong testing in internal network channel through the 1Gb Ethernet NIC device. The NetPIPE performance of the physical platform attains 718.45Mbps (as NetPIPE_switch) and the performance without the Ethernet switch device can achieve 765.40Mbps (as NetPIPE_noswitch) that is faster than forwarding all packages through 1Gb switch device. For Xen kernel environment, the maximal value of throughput using NetPIPE program is 580.21 Mbps, and when testing without the Ethernet switch device that can obtain 651.94Mbps correspondingly. Generally speaking, the network communication performance for virtual cluster platforms are restricted at lower bandwidth than the physical system with virtual network device for Xen PV, Xen FV and KVM FV, which are attained 356.32Mbps (374.81), 71.66Mbps (82.76) and 174.62Mbps (175.62) respectively. Clearly, the worst is Xen FV among all virtual platforms.

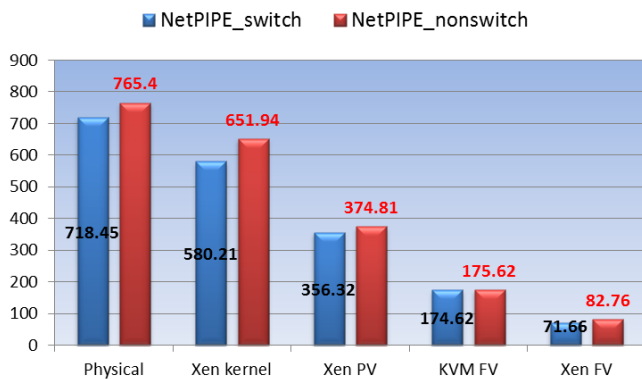


Figure 3. The NetPIPE performance.

Figure 4 is the Radar chart that indicates significance experiment results of MPI mode on whole system via HPC benchmark including PTRANS, MPIFTT, Communication bandwidth and latency. For PTRANS, The physical platform can get 0.402 GBs, Xen FV shows worst that is around 0.045 GBs barely. Using FTT to measures the floating point rate

with MPI can achieve the best result 1.301 Gflops in physical cluster platform, and the worst result is only 0.118 Gflops likes Xen FV platform. For bandwidth and latency under varying conditions of internal network between multiple pairs of nodes, physical platform get 0.009 GBytes of bandwidth and 56.72 usec of latency. Related network evaluation consist of bandwidth and latency for Xen PV, KVM FV, Xen FV, which are attained following 0.008GBytes (142.08 usec), 0.003GBytes (609.90 usec) and 0.001GBytes (952.36 usec) separately. That express the Xen PV platform has better communication performance for all testing virtual platforms.

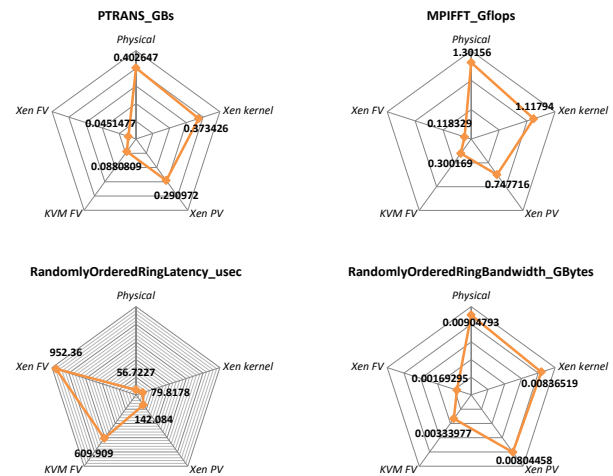


Figure 4. Significant results by HPC benchmark.

Formula (3) shows the based principle of overhead between physical and virtual platforms. Rmax_p represents the Rmax of physical platform, and Rmax_v is the Rmax of virtual platform. Thus, the overhead value can explain the reduced level of performance, while value close to 0 means the virtual platform is as well as the physical platform. On the other hand, if the value is approaching 100%, it means the performance of the measured virtual platform is very bad.

$$Overhead = \frac{R_{max_p} - R_{max_v}}{R_{max_p}} \times 100\% \quad (3)$$

Figure 5 illustrates the map between Efficiency (as red line) and overhead (as blue line) in each platforms, the overhead of Xen kernel environment is approximately 0.58%, the best performance next to the physical platform. The virtual platform with Xen PV has a quite low overhead at 7.04%, and KVM FV has a high overhead at 45.82%. Xen FV has 71.35% overhead that is clearly the worst virtualization platform for correlate computing and communication performance.

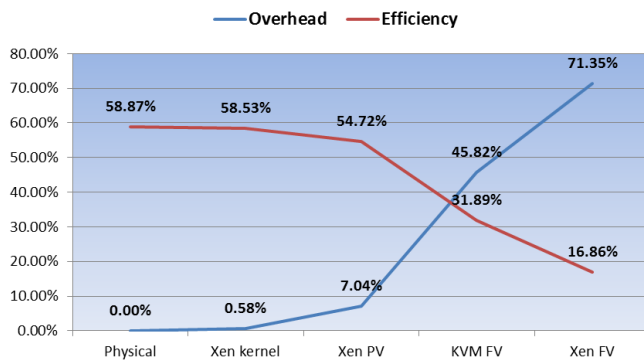


Figure 5. Overhead and Efficiency.

With related experiment results of this section, we can rank the whole performance of testing platforms in the following order: physical platform, Xen kernel, Xen PV, KVM FV and Xen FV.

5 Conclusions

In our paper, standard benchmark tools including HPCC benchmark and NetPIPE are adopted to estimate computing performance among physical platform, Xen kernel platform, Xen PV platform, Xen FV platforms and KVM FV platform. Based on significant experiment results, by comparing the virtualization performance of Xen PV platform is better than other virtual platforms through FV of KVM and Xen, with the latter being the worst virtualization performance like cpu, memory and network in all testing platforms. Hence, if virtual cluster system or cloud platform was built using PV technology of Xen, it could be support effective virtualization performance in computing and communication. However, there is a troublesome disadvantage that the PV mechanism can't support user to construct different virtual OS and Guest OS on host machine. Although virtual platforms using KVM FV technology shows the average efficiency via HPCC benchmark tool and NetPIPE program, nonetheless it is able to deploy on many different types of virtual platforms for novel and popular cloud service.

6 Acknowledgment

This work is supported by National Science Council, R.O.C., under the contract number of NSC NSC99-2218-E492-00.

7 References

- [1] AMD-V, <http://sites.amd.com/us/business/it-solutions/virtualization/Pages/virtualization.aspx>
- [2] b_eff, https://fs.hlrs.de/projects/par/mpi/b_eff/
- [3] FFT, <http://www.ffte.jp/>
- [4] HPCC, <http://icl.cs.utk.edu/hpcc/>
- [5] HPL, <http://www.netlib.org/benchmark/hpl/>
- [6] Intel-VT, <http://www.intel.com/technology/virtualization/>
- [7] KVM, <http://www.linux-kvm.org/>
- [8] Microsoft Virtual PC, <http://www.microsoft.com/windows/virtual-pc/>
- [9] NetPIPE, <http://www.scl.ameslab.gov/netpipe/>
- [10] PTRANS, <http://www.netlib.org/parkbench/>
- [11] RandomAccess, <http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/>
- [12] STREAM, <http://www.cs.virginia.edu/stream/>
- [13] TOP500, <http://www.top500.org/>
- [14] VMware, <http://www.vmware.com/>
- [15] Xen, <http://www.xen.org/>
- [16] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin and Anthony Liguori, "kvm: the Linux Virtual Machine Monitor," *In Proceedings of the Linux Symposium*, vol. 1, pp. 225-230, June 2007.
- [17] Carl J. Young, "Extended Architecture and Hypervisor Performance," *Proceedings ACM SIGARCH-SIGOPS Workshop on Virtual Computer Systems*, Cambridge, MA, 1973.
- [18] Danhua Guo, Guangdeng Liao and L. N. Bhuyan, "Performance Characterization and Cache-Aware Core Scheduling in a Virtualized Multi-Core Server under 10GbE," *Workload Characterization, 2009. IISWC 2009*, pp. 168-177, Oct. 2009.
- [19] Greg Goth, "Virtualization: Old Technology Offers Huge New Potential," *IEEE Distributed Systems Online*, vol. 8, no. 2, 2007.
- [20] Gerald J. Popek and Robert P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM*, vol. 17, pp. 412-421, July 1974.
- [21] G. Neiger, A. Santoni et al, "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization", *Intel Computer Journal*, vol. 10, issue 3, August 2006.
- [22] I. Habib, "Virtualization with KVM," *Linux Journal*, Vol. 2008, Feb. 2008.
- [23] J. J. Dongarra, P. Luszczyk, and A. Petitet, "The LINPACK benchmark: past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803-820, 2003.
- [24] J. P. Buzen and U. O. Gagliardi, "The Evolution of Virtual Machine Architecture," *National Computer Conference Proceedings, AFIPS Press*, vol. 42, pp. 291-299, June 1973.
- [25] L. Nussbaum, F. Anhalt, O. Mornard and J.-P. Gelas, "Linux-based virtualization for HPC clusters," *Linux Symposium*, pp. 221-234, July 2009.
- [26] M. Fenn, M. Murphy, and S. Goasguen, "A Study of a KVM-based Cluster for Grid Computing," *47th ACM Southeast Conference*, March 2009.
- [27] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebarger, Ian Pratt and Andrew Warfield, "Xen and the Art of Virtualization," *Proceedings of the 2003 Symposium on Operating Systems Principles*, October 2003.
- [28] Q.O. Snell, A.R. Mikler and J.L. Gustafson, "NetPIPE: A Network Protocol Independent Performance Evaluator," *Ames Laboratory / Scalable Computing Lab, Iowa State.*, 1997.
- [29] R. A. Meyer and L. H. Seawright, "A Virtual Machine Time-Sharing System," *IBM Systems Journal*, vol. 9, no. 3, 1970.
- [30] R. P. Goldberg, "Architecture of Virtual Machines," *National Computer Conference Proceedings, AFIPS Press*, vol. 42, pp. 309-318, June 1973.
- [31] R. P. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer*, vol. 7, no. 6, pp. 34-45, June 1974.

- [32] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, Larry Smith, "Intel Virtualization Technology," *IEEE Computer*, vol. 38, no. 5, pp. 48-56, May 2005.
- [33] Thomas C. Bressoud and Fred B. Schneider, "Hypervisor-based Fault Tolerance," *ACM Transactions on Computer Systems*, vol. 14, pp. 80-107, 1996.
- [34] Wei Chen, Hongyi Lu, Li Shen, Zhiying Wang, Nong Xiao and Dan Chen, "A Novel Hardware Assisted Full Virtualization Technique," *The 9th International Conference for Young Computer Scientists*, pp. 1292-1297, Nov. 2008.
- [35] Yan Wen, Jinjing Zhao, Huaimin Wang and Jiannong Cao, "Implicit Detection of Hidden Processes with a Feather-Weight Hardware-Assisted Virtual Machine," *ACISP '08 Proceedings of the 13th Australasian conference on Information Security and Privacy*, pp. 361-375, 2008.

Market Basket Analysis Algorithm with Map/Reduce of Cloud Computing

Jongwook Woo

Computer Information Systems Department
California State University
Los Angeles, CA

Yuhang Xu

Computer Information Systems Department
California State University
Los Angeles, CA

Abstract – *Map/Reduce approach has been popular in order to compute huge volumes of data since Google implemented its platform on Google Distributed File Systems (GFS) and then Amazon Web Service (AWS) provides its services with Apache Hadoop platform. Map/Reduce motivates to redesign and convert the existing sequential algorithms to Map/Reduce algorithms for big data so that the paper presents Market Basket Analysis algorithm with Map/Reduce, one of popular data mining algorithms. The algorithm is to sort data set and to convert it to (key, value) pair to fit with Map/Reduce. It is executed on Amazon EC2 Map/Reduce platform. The experimental results show that the code with Map/Reduce increases the performance as adding more nodes but at a certain point, there is a bottle-neck that does not allow the performance gain. It is believed that the operations of distributing, aggregating, and reducing data in Map/Reduce should cause the bottle-neck.*

Keywords: Map/Reduce, Market Basket Analysis, Data Mining, Association Rule, Hadoop, Cloud Computing

1 Introduction

Before Internet and Web did not exist, we did not have enough data so that it was not easy to analyze people, society, and science etc with the limited volumes of data. Contradicting to the past, after Internet and web, it has been more difficult to analyze data because of its huge volumes, that is, tera- or peta-bytes of data. Google faced to the issue as she collected big data and the existing file systems were not sufficient to handle the data efficiently. Besides, the legacy computing power and platforms were not useful for the big data. Thus, she implemented Google File Systems (GFS) and Map/Reduce parallel computing platform, which Apache Hadoop project is motivated from.

Hadoop is the parallel programming platform built on Hadoop Distributed File Systems (HDFS) for Map/Reduce computation that processes data as (key, value) pairs. Hadoop has been receiving highlights for the enterprise computing because business world always has the big data such as log files for web transactions. Hadoop is useful to process such big data for business intelligence so that it has been used in data mining for past few years. The era of

Hadoop means that the legacy algorithms for sequential computing need to be redesigned or converted to Map/Reduce algorithms. Therefore, in this paper, a Market Basket Analysis algorithm in data mining with Map/Reduce is proposed with its experimental result in Elastic Compute Cloud (EC2) and (Simple Storage Service) S3 of Amazon Web Service (AWS).

People have talked about Cloud Computing that is nothing else but the services we have used for several years: hosting service, web email service, document sharing service, and map API service etc. It is categorized into Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). SaaS is to use a service via Internet without installing or maintaining the software, for example, web email services. PaaS is to have a computing or storage service without purchasing hardware or software, for example, hosting services. IaaS is to have utility computing service that is similar to SaaS but to purchase only the amount of time to use the service like AWS [6, 7]. AWS provides S3, EC2, and Elastic MapReduce services for Map/Reduce computation as IaaS and SaaS in cloud computing.

In this paper, section 2 is related work. Section 3 describes Map/Reduce and Hadoop as well as other related projects. Section 4 presents the proposed Map/Reduce algorithm for Market Basket Analysis. Section 5 shows the experimental result. Finally, section 6 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 and many data mining books illustrate 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.

As far as we understand, there is not any other to present Market Basket Analysis algorithms with Map/Reduce. The approach in the paper is to propose the

algorithm and to convert data to (key, value) pair and execute the code on Map/Reduce platform.

3 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 huge volumes of data set 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.

3.1 Map/Reduce in parallel computing

Map/Reduce programming platform is implemented in the Apache Hadoop project that develops open-source software for reliable, scalable, and distributed computing. Hadoop can compose hundreds of nodes that process and compute peta- or tera-bytes of data working together. Hadoop was inspired by Google's MapReduce and GFS as Google has had needs to process huge data set for information retrieval and analysis [1]. It is used by a global community of contributors such as Yahoo, Facebook, and Twitters. Hadoop's subprojects include Hadoop Common, HDFS, MapReduce, Avro, Chukwa, HBase, Hive, Mahout, Pig, and ZooKeeper etc [2].

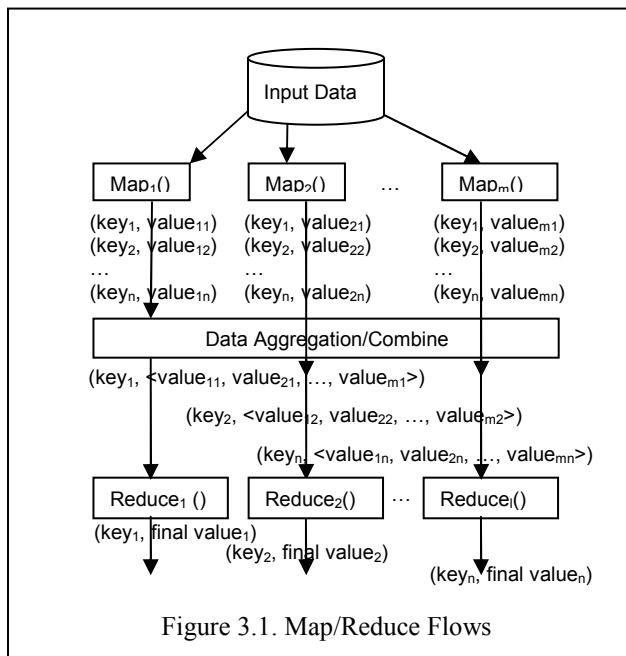


Figure 3.1. Map/Reduce Flows

The map and reduce functions run on distributed nodes in parallel. Each map operation can be processed independently on each node and all the operations can be performed in parallel. But in practice, it is limited by the data source and/or the number of CPUs near that data. The reduce functions are in the similar situation because they are from all the output of the map operations. However, Map/Reduce can handle significantly huge data sets since

data are distributed on HDFS 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 parallelly computes and generates the list of (key, value) as output on map/reduce functions. In map function, the master node parts the input into smaller sub-problems, and distributes those to worker nodes. Those worker nodes process smaller problems, and pass the answers back to their master node. That is, map function takes inputs $(k1, v1)$ and generates $\langle k2, v2 \rangle$ where $\langle \rangle$ represents list or set. Between map and reduce, there is a combiner that resides on map node, which takes inputs $(k2, \langle v2 \rangle)$ and generates $\langle k2, v2 \rangle$.

In reduce function, the master node takes the answers to all the sub-problems and combines them in some way to get the output, the answer to the problem [1, 2]. That is, reduce function takes inputs $(k2, \langle v2 \rangle)$ and generates $\langle k3, v3 \rangle$. Figure 3.1 illustrates Map/Reduce control flow where each $value_{mn}$ is simply 1 and gets accumulated for the occurrence of items together in the proposed Market Basket Analysis Algorithm.

3.2 Database for Big Data

Input/Output files are processed on HDFS instead of using HBase DB in the paper. However, as HBase is interesting and will be integrated with the algorithm in the future, the section briefly introduces HBase.

There are some drawbacks when we use RDBMS to handle huge volumes of data, like impossible deleting, slow inserting, and random failing. HBase on HDFS is distributed database that supports structured data storage for horizontally scalable tables. It is column oriented semi-structured data store.

It is relatively easy to integrate with Hadoop Map/Reduce because HBase consists of a core map that is composed of keys and values - each key is associated with a value. Users store data rows in labeled tables. A data row has a sortable key and an arbitrary number of columns. The table is stored sparsely, so that rows in the same table can have different columns.

Using the legacy programming languages such as Java, PHP, and Ruby, we can put data in the map as Java JDBC does for RDBMS. The file storage of HBase can be distributed over an array of independent nodes because it is built on HDFS. Data is replicated across some participating nodes. When the table is created, the table's column families are generated at the same time. We can retrieve data from HBase with the full column name in a certain form. And then HBase returns the result according to the given queries as SQL does in RDBMS [10].

3.3 The Issues of Map/Reduce

Although there are advantages of Map/Reduce, for some researchers and educators, it is:

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. Not new at all - it represents a specific implementation of well known techniques developed tens of years ago, especially in Artificial Intelligence
4. Data should be converted to the format of (key, value) pair for Map/Reduce, which misses most of the features that are routinely included in current DBMS
5. Incompatible with all of the tools or algorithms that have been built [4].

However, the issues clearly show us not only the problems but also the opportunity where we can implement algorithms with Map/Reduce approach, especially for big data set. It will give us the chance to develop new systems and evolve IT in parallel computing environment. It started a few years ago and many IT departments of companies have been moving to Map/Reduce approach in the states.

4 Market Basket Analysis Algorithm

Market Basket Analysis is one of the Data Mining approaches to analyze the association of data set. The basic idea is to find the associated pairs of items in a store when there are transaction data sets as in Figure 4.1.

If store owners list a pair of items that are frequently occurred, s/he could control the stocks more intelligently, to arrange items on shelves and to promote items together etc. Thus, s/he should have much better opportunity to make a profit by controlling the order of products and marketing.

Transaction 1: cracker, icecream, beer
 Transaction 2: chicken, pizza, coke, bread
 Transaction 3: baguette, soda, hering, cracker, beer
 Transaction 4: bourbon, coke, turkey
 Transaction 5: sardines, beer, chicken, coke
 Transaction 6: apples, peppers, avocado, steak
 Transaction 7: sardines, apples, peppers, avocado, steak
 ...

Figure 4.1 Transaction data at a store

Total number of Items: 322,322
 Ten most frequent Items:

cracker, beer	6,836
artichok, avocado	5,624
avocado, baguette	5,337
bourbon, cracker	5,299
baguette, beer	5,003
corned, hering	4,664
beer, hering	4,566
...	

Figure 4.2 Top 10 pair of items frequently occurred at store

For example, people have built and run Market Basket Analysis codes – sequential codes - that compute the top 10 frequently occurred pair of transactions as in Figure 4.2. At the store, when customers buy a cracker, they purchase a beer as well, which happens 6,836 times and bourbon as well in 5,299 times. Thus, the owner can refer to the data to run the store.

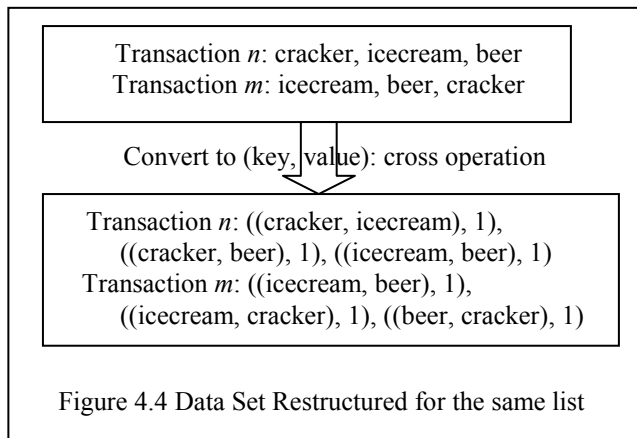
4.1 Data Structure and Conversion

The data in Figure 4.1 is composed of the list of transactions with its transaction number and the list of products. For Map/Reduce operation, the data set should be structured with (key, value) pairs. The simplest way used in the paper is to pair the items as a key and the number of key occurrences as its value in the basket, especially for all transactions, without the transaction numbers. Thus, Figure 4.1 can be restructured as Figure 4.3 assuming collecting a pairs of items in order 2 – two items as a key.

```
< (cracker, icecream), (cracker, beer) >
< (chicken, pizza), (chicken, coke), (chicken, bread) >
< (baguette, soda), (baguette, hering), (baguette, cracker), (baguette, beer) >
< (bourbon, coke), (bourbon, turkey) >
< (sardines, beer), (sardines, chicken), (sardines, coke) >
>
```

Figure 4.3 Data Set restructured for Map/Reduce

However, if we select the two items in a basket as a key, there should be incorrect counting for the occurrence of the items in the pairs. As shown in Figure 4.4, transactions *n* and *m* have the items (cracker, icecream, beer) and (icecream, beer, cracker), which have the same items but in different order.

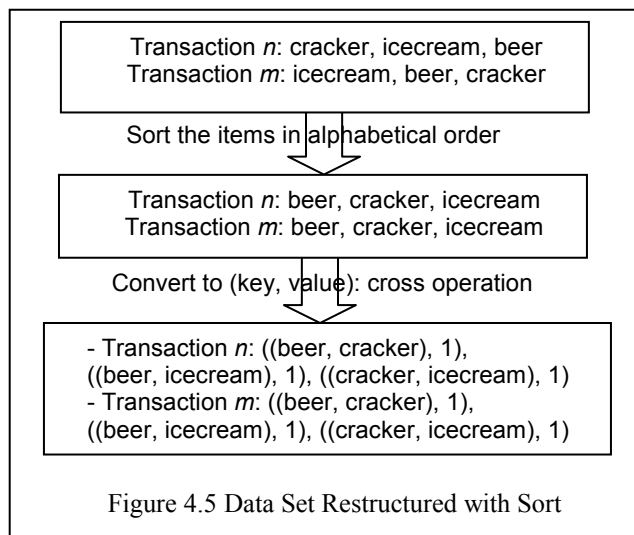


That is, for (cracker, icecream, beer), the possible pair of items in (key, value) are ((cracker, icecream), 1), ((cracker, beer), 1), ((icecream, beer), 1). And, for (icecream, beer, cracker), the possible pair of items are

((icecream, beer), 1), ((icecream, cracker), 1), ((beer, cracker), 1).

Therefore, we have total SIX different pair of items that occurs only once respectively, which should be THREE different pairs. That is, keys (cracker, icecream) and (icecream, cracker) are not same even though they are, which is not correct.

We can avoid this issue if we sort the transaction in alphabetical order before generating (key, value) as shown in Figure 4.5. Now each transaction have the following THREE pair of items ((beer, cracker), 1), ((beer, icecream), 1), ((cracker, icecream), 1). That is TWO different pair of items that occurs twice respectively so that we accumulate the value of the occurrence for these two transactions as follows: ((beer, cracker), 2), ((beer, icecream), 2), ((cracker, icecream), 2), which is correct to count the total number of occurrences.



4.2 The algorithm

The Market Basket Analysis (MBA) Algorithms for Mapper and Reducer are illustrated in Figures 4.6 and 4.7 respectively. Mapper reads the input data and creates a list of items for each transaction. As a mapper of a node reads each transaction on Hadoop, it assigns mappers to number of nodes, where the assigning operation in Hadoop is hidden to us. For each transaction, its time complexity is $O(n)$ where n is the number of items for a transaction.

Then, the items in the list are sorted to avoid the duplicated keys as shown in Figures 4.4 and 4.5. Its time complexity is $O(n \log n)$ on merge sort. Then, the sorted items should be converted to pairs of items as keys, which is a cross operation in order to generate cross pairs of the items in the list as shown in Figures 4.4 and 4.5. Its time complexity is $O(n \times m)$ where m is the number of pairs that occurs together in the transaction. Thus, the time complexity of each mapper is $O(n + n \log n + n \times m)$.

- 1: Reads each transaction of input file and generates the data set of the items:
 $\langle V_1 \rangle, \langle V_2 \rangle, \dots, \langle V_n \rangle$ where $\langle V_n \rangle: (v_{n1}, v_{n2}, \dots, v_{nm})$
- 2: Sort all data set $\langle V_n \rangle$ and generates sorted data set $\langle U_n \rangle$:
 $\langle U_1 \rangle, \langle U_2 \rangle, \dots, \langle U_n \rangle$ where $\langle U_n \rangle: (u_{n1}, u_{n2}, \dots, u_{nm})$
- 3: Loop While $\langle U_n \rangle$ has the next element;
note: each list U_n is handled individually
 - 3.1: Loop For each item from u_{n1} to u_{nm} of $\langle U_n \rangle$ with NUM_OF_PAIRS
 - 3.a: generate the data set $\langle Y_n \rangle: (y_{n1}, y_{n2}, \dots, y_{ni})$;
 $y_{ni}: (u_{nx}, u_{ny})$ is the list of self-crossed pairs of $(u_{n1}, u_{n2}, \dots, u_{nm})$ where $u_{nx} \neq u_{ny}$
 - 3.b: increment the occurrence of y_{ni} ;
note: (key, value) = (y_{ni} , number of occurrences)
 - 3.2: End Loop For
4. End Loop While
5. Data set is created as input of Reducer: (key, <value>) = (y_{ni} , <number of occurrences>)

Figure 4.6. MBA Algorithm for Mapper

The reducer is to accumulate the number of values per key. Thus, its time complexity is $O(v)$ where v is the number of values per key.

- 1: Read (y_{ni} , <number of occurrences>) data from multiple nodes
2. Add the values for y_{ni} to have (y_{ni} , total number of occurrences)

Figure 4.7. MBA Algorithm for Reducer

4.3 The code

The *ItemCount.java* code is implemented on Hadoop 0.20.2 and 0.21.0 and executable on stand-alone and clustered modes. The code generates the top 10 associated items that customers purchased together as shown in Figure 4.2. Anyone can download the files to test it, which takes the sample input "AssociationsSP.txt" as introduced in the blog [8]. The sample input has 1,000 transactions with data as shown in Figure 4.1.

5 Experimental Result

We have 5 transaction files for the experiment: 400 MB (6.7M transactions), 800MB (13M transactions), 1.6 GB (26M transactions). Those are run on small instances of AWS EC2 which allows to instantiate number of nodes requested, where each node is of 1.0-1.2 GHz 2007 Opteron or Xeon Processor, 1.7GB memory, 160GB storage on 32 bits platform. The data are executed on 2, 5, 10, 15, and 20

nodes respectively and its execution times are shown in Table 5.1. For 13 and 26 Mega transactions of 2 nodes, it took too long to measure the execution times so that we do not execute them and its times are Not Applicable (NA) in the Table 5.1.

Trax Nodes	6.7M (400MB)	13M (800MB)	26M (1.6GB)
2	9,133	NA	NA
5	5,544	8,717	15,963
10	2,910	5,998	8,845
15	2,792	2,917	5,898
20	2,868	2,911	5,671

Table 5.1. Execution time (sec) at Map Task:

The output of the computation in Table 5.2 presents the number of items (total: 1.3G) and keys (total: 212) that are associated pair of items in order 2, especially for data of 26M transactions in file size 1.6GB. And, the 10 most frequently occurred items and its frequency, which is (key, value), are shown.

Total number of keys in order 2: 212	
Total number of items: 1,255,922,927	
Items Paired (key)	Frequency (value)
cracker, heineken	208,816,643
artichok, avocado	171,794,426
avocado, baguette	163,027,463
bourbon, cracker	161,866,763
baguette, heineken	152,824,775
corned_b, hering	142,469,636
heineken, hering	139,475,906
bourbon, heineken	126,310,383
baguette, cracker	125,699,308
artichok, heineken	125,180,072

Table 5.2. 10 most frequently associated items on 1.6GB 26M transactions

Figure 5.1 based on Table 5.1 is the chart of the experimental result with 400 MB (6.7M transactions), 800MB (13M transactions), 1.6 GB (1,600 MB 26M transactions). The more the nodes are, the faster the computation times are. Since the algorithm is simply to sort the data set and then convert it to (key, value) pairs, the linear result is expected. The performance is linearly increased by some nodes for some transaction data sets but it has the limitation. For 400MB file, there is not much difference among nodes 10, 15 and 20. Similarly, for 800MB and 1.6GB files, there are not many differences between nodes 15 and 20. There is bottleneck in EC2 small instance, which shows that there is a trade-off between the number of nodes and the operations of distributing

transactions data to nodes, aggregating the data, and reducing the output data for each key so that it should not have much performance gain even though adding more nodes for faster parallel computation.

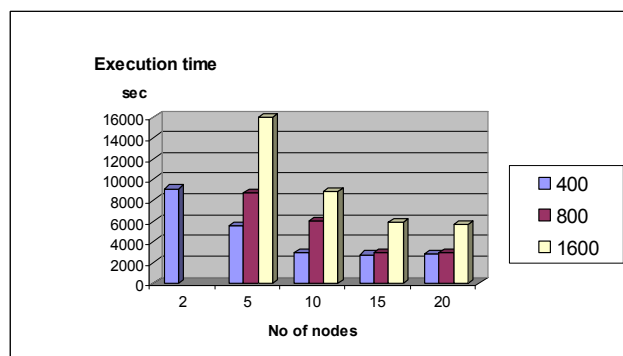


Figure 5.1. Chart for Execution Time

In summary, the experimental data illustrates that even though the number of nodes are added, at a certain point, there is a bottleneck that cannot increase the performance because of the time to distribute, aggregate, and reduce the data to Map/Reduce nodes.

6 Conclusion

Hadoop with Map/Reduce motivates the needs to propose new algorithms for the existing applications that have had algorithms for sequential computation. Besides, it is (key, value) based restricted parallel computing so that the legacy parallel algorithms need to be redesigned with Map/Reduce.

In the paper, the Market Basket Analysis Algorithm on Map/Reduce is presented, which is association based data mining analysis to find the most frequently occurred pair of products in baskets at a store. The data set shows that associated items can be paired with Map/Reduce approach. Once we have the paired items, it can be used for more studies by statically analyzing them even sequentially, which is beyond this paper.

The algorithm has been executed on EC2 small instances of AWS with nodes 2, 5, 10, 15, and 20. The execution times of the experiments show that the proposed algorithm gets better performance while running on large number of nodes to a certain point. However, from a certain point, Map/Reduce does not guarantee to increase the performance even though we add more nodes because there is a bottle-neck for distributing, aggregating, and reducing the data set among nodes against computing powers of additional nodes.

7 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 , O'Reilly 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] “ Introduction to Cloud Computing”, Jongwook Woo, the 10th KOCSEA 2009 Symposium, UNLV, Dec 18-19, 2009
- [7] “The Technical Demand of Cloud Computing”, Jongwook Woo, Korean Technical Report of KISTI (Korea Institute of Science and Technical Information), Feb 2011
- [8] “Market Basket Analysis Example in Hadoop, <http://dal-cloudcomputing.blogspot.com/2011/03/market-basket-analysis-example-in.html>”, Jongwook Woo, March 2011
- [9] “SQL MapReduce framework ”, Aster Data, <http://www.asterdata.com/product/advanced-analytics.php>
- [10] Apache HBase, “<http://hbase.apache.org/>”
- [11] “Data-Intensive Text Processing with MapReduce”, Jimmy Lin and Chris Dyer, Morgan & Claypool Publishers, 2010.
- [12] GNU Coord, <http://www.coordguru.com/>
- [13] “Integrated Information Systems Architecture in e-Business”, Jongwook Woo, Dong-Yon Kim, Wonhong Cho, MinSeok Jang, The 2007 international Conference on e-Learning, e-Business, Enterprise Information Systems, e-Government, and Outsourcing, Las Vegas (June 26-29, 2007)

SESSION
PARALLEL ALGORITHMS AND APPLICATIONS

Chair(s)

TBA

Graph Generation on GPUs using Dynamic Memory Allocation

A. Leist and K.A. Hawick

¹Computer Science, Massey University

²Albany, North Shore 102-904, Auckland, New Zealand

Abstract—Complex networks are often studied using statistical measurements over many independently generated samples. Irregular data structures such as graphs that involve dynamical memory management and “pointer chasing” are an important class of application and have attracted recent interest in the form of the Graph500 benchmark formulation. The generation of simulated sample network graphs and measurement of their properties can be accelerated using Graphical Processing Units (GPUs) and we discuss some algorithmic approaches using Compute Unified Device Architecture (CUDA). We particularly discuss recent support for dynamic memory allocation within CUDA GPU code and present some performance data for Watts’ α small-world network model.

Keywords: CUDA; GPGPU; graph generation; dynamic memory allocation

1. Introduction

Generating synthetic graph or network data is very useful in investigating the statistical properties of various models. Properties such as the average path length [1], [2], clustering coefficient [3]; circuit and loop structure [4]; between-ness and reachability of the networks defined by various models can all be measured quantifiably on various sample network realisations. Scaling can be studied as the size of the generated synthetic data set is varied and for some models other parameters can also be varied and their effect systematically studied. This is a powerful approach as real data from experiments, real physical systems or surveys often contain noise or inaccuracies whereas a comparison with synthetic data – particularly when averaged over many samples – can make it a lot easier to identify effects and make comparisons with theoretical predictions.

A great deal of work has been reported in the literature on the properties of complex networks [5], [6]. These include: classic random graphs [7], [8]; small-World graphs such as the Watts-Strogatz β -model [9] and scale-free and preferential attachment models such as that of Barabási and Albert [10]. Applications include social problems like collaboration networks [11]; clustering and community structure determination [12]; and the Internet [13] and world-wide web [14].

An interesting model that has attracted somewhat less effort is Watts’ α -model. This sociological model presents some challenging computational problems in generating large synthetic sample networks. A sample network generated with the α -model is illustrated in Figure 1.

Investigation of the properties of small-world networks [15], [16] reveals that many of the interesting phenomena are only revealed over size scales that vary with some power law. Consequently it is necessary to generate networks with a large number of nodes with potentially large numbers of edges as well. This is computationally challenging and parallel computing techniques become necessary just to generate good sample sizes of large synthetic networks.

Parallel graph generation has attracted recent interest due to the announcement of the Graph500 benchmark [17] for supercomputer systems in 2010. Led by Murphy at the Sandia National Laboratory, the Graph500 is an important attempt to recognise that many of the application problems that are run on supercomputers are not necessarily well characterised by the linear algebra problem represented by the widely quoted Linpack benchmark, which is used to compile the Top500 [18] list of worldwide supercomputers.

The Graph500 benchmark is still in an early stage of uptake and adoption and consists of a two-part benchmark to generate a scale-free graph and to perform parallel searches upon it. The synthetic graphs can be quite large in size and the scalability of performance with graph size is an important aspect of this sort of performance benchmark.

We have chosen to study algorithms for generation of Watts α -model networks on data-parallel accelerator devices such as graphical processing units (GPUs). There has been a great deal of interest in the performance and capabilities of GPUs in recent years and several of the top performing supercomputers in the Top500 list in fact employ GPU accelerators to attain their performance ratings with Linpack. It therefore is of topical interest to explore the performance of GPUs in carrying out graph generation and in particular to determine how well some recent features of GPU programming models and systems cope with these tasks.

We explore the performance of GPUs using NVIDIA’s Compute Unified Device Architecture (CUDA) and in this paper we look at a recently introduced CUDA capability – to allocate memory dynamically within the GPU code itself. The programming model for GPUs has until recently been that of calling compute kernels written in CUDA to run on the GPU as “pseudo subroutines” from the conventional CPU host program. Memory management had to be performed in the CPU calling code only. The recent introduction [19] of the `malloc` and `free` routines into CUDA offers some new possibilities for dynamical memory “pointer-chasing” applications such as graph generation.

Our article is structured as follows. In Section 2 we define the Watts α -model and summarise the generation algorithm for synthetic networks of this type. The relevant features of the CUDA GPU architecture are described in Section 3. In Section 4 we expand on details of our serial and parallel implementations of the model using single and multi-core CPU approaches as well as a GPU implementation. We present some scaling and performance results in Section 5 and a discussion of their implications in Section 6. We offer some conclusions and areas for further work with multi-core and GPU graph generation algorithms in Section 7.

2. The α -Model

Watts introduced the α -model as a way of interpolating between the two extreme social network cases of the “cave-man” model and “solaris” model [9]. The “cave-man” model

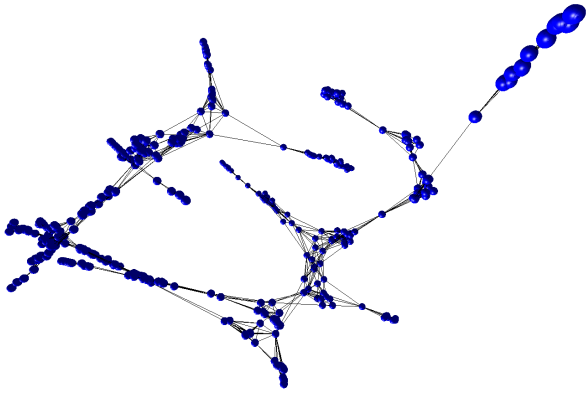


Fig. 1: Graph generated using the Watts α -model with $n = 500$ vertices, degree $k = 10$ and $\alpha = 2.0$. Larger values of α increase the number of random edges and therefore the connectedness of the graph.

posits a situation where individuals (nodes in the network) may be well connected (via a graph edge) within their local “cave” of friends, but where there are no connections between “caves.” The “solaris” model is the random graph case where relationship edges are plentiful and global but random. The α model introduces a parameter (α) that interpolates between these situations.

Watts defines the α -model [20] for choices of number of nodes n , connectivity k and parameter α as follows:

- 1) Consider in turn each vertex i . Vertices $i = 1 \dots n$ are chosen in random order, but once a vertex has been wired by choosing a new neighbour, it may not choose again until all other vertices have taken their turn at this step.
- 2) For every other vertex $j \neq i$, compute $R_{i,j}$ according to Equation 1, imposing the additional constraint that $R_{i,j} = 0$ if vertices i and j are already connected.
- 3) Then sum the $R_{i,j}$ over all $j \neq i$ and normalise each to obtain variables $P_{i,j} = R_{i,j} / \sum_{l \neq i} R_{i,l}$. Now $\sum_j P_{i,j} = 1$, $P_{i,j}$ can be interpreted as the probability that i will be connected to j . In addition, $P_{i,j}$ can be interpreted geometrically as dividing $[0, 1)$ – the unit interval – into $n - 1$ half-open subintervals with length $P_{i,j}, \forall j \neq i$.
- 4) A uniform pseudo-random variable is generated on $[0, 1)$. It will fall into one of the subintervals, which we identify as corresponding to j_* .
- 5) We then connect i to j_* .

The five steps of this procedure are repeated until the predetermined number of edges ($M = (k * n) / 2$) has been constructed.

$$R_{i,j} = \left\{ \begin{array}{ll} 1, & m_{i,j} \geq k \\ \left[\frac{m_{i,j}}{k} \right]^\alpha (1-p) + p, & k > m_{i,j} > 0 \\ p, & m_{i,j} = 0 \end{array} \right\} \quad (1)$$

where:

$R_{i,j}$ = a measure of vertex i 's propensity to connect to vertex j (zero if they are already connected)
 $m_{i,j}$ = number of vertices adjacent to both i and j
 k = the average degree of the graph
 p = a baseline, random probability of an edge (i, j) existing ($p \ll \binom{n}{2}^{-1}$)

α = a tunable parameter, $0 \leq \alpha \leq \infty$

In the cases of $\alpha \equiv 0$ we retain the cave-man clumped model with disconnects. In the case of $\alpha \rightarrow \infty$ we recover the case of the random graph model. The α model exhibits a phase change in its properties at a particular value of the α parameter. For example the average path-length between all possible pairs of nodes in the graph is an interesting metric. At low α values it starts low and rises to a peak with increasing α , then falls away to a flat fixed value at high α [9].

3. NVIDIA GPU Architecture

GPU architectural details have been described in detail elsewhere [21] but for completeness we give here a brief summary of memory and other architectural features critical to an explanation of our graph generation algorithm.

Of the many GPGPU APIs available [19], [22], [23], [24], NVIDIA's CUDA stands out as the most developed and advanced API. However, it can only be used in conjunction with NVIDIA GPUs. Our development of GPGPU applications uses the CUDA toolkit and is thus limited to graphics hardware from the same manufacturer. While we specifically discuss the NVIDIA device architecture, most of the concepts are transferable to products from other vendors.

CUDA compliant GPUs contain a scalable array of Streaming Multiprocessors (SMs), which in turn are host to up to 32 Scalar Processors (SPs) in the latest generation of “Fermi”-architecture based devices. Each SM can perform computation independently but the SP cores within the same multiprocessor all execute instructions synchronously. NVIDIA call this paradigm Single Instruction Multiple Thread (SIMT) [19].

The GPU hardware is capable of managing thousands of threads at a time. It performs the tasks of creating and scheduling threads in hardware, which keeps the overhead of managing a large number of threads very small. The GPU is designed to support fine-grained parallelism. In order to utilise the hardware properly, applications need to split the work load into many threads, each of which performs a relatively small unit of work.

GPUs contain several different types of memory. The latest generation of Fermi devices even adds an implicit L1/L2 cache hierarchy to the mix, a feature previously not available on GPUs. The GPU memory types explicitly accessible from CUDA are as follows [19]: *Registers* are used to store local variables belonging to each thread. When registers can not hold all local variables, then they spill into *Local Memory*, which is an area of the L1/L2 caches on Fermi devices and a section of much slower device memory on older GPUs. *Global Memory* is the largest section of memory and also resides in device memory. It is the only memory type that the CPU can read from and write to. It is accessible by all threads, but has the slowest access times (400-600 clock cycles). The total transaction time can be improved by coalescing – 16 sequential threads that access 16 sequential and word-aligned memory addresses can coalesce the read/write into a single memory transaction. *Shared Memory* is a fast on-chip read/write memory that can be used as an explicit cache and to share data between threads in the same thread block. Provided that threads access different memory banks (16 for pre-Fermi devices and 32 for Fermi devices), memory transactions are as fast as accessing registers. *Texture Memory* is a cached method for reading from a section of global

memory that is bound to a texture reference. Texture memory is optimised for spatial memory access. It performs best when streaming data that is located spatially close in the dimensions defined by the texture reference. *Constant Memory* is another cached method for reading from global memory. A global memory read is only required in the event of a cache miss. If all threads read the same value from the constant cache, the cost is the same as reading from registers.

The developer must explicitly utilise the memory types best suited for the given task. Because bandwidth is limited, the optimal memory access strategy can provide huge performance gains, and the wrong strategy can severely hamper performance.

A CUDA application has a number of threads organised into thread blocks, which are all arranged into a grid. A block of threads has up to three dimensions (x,y,z) and can contain no more than 512 (1024 on Fermi devices) threads. A grid has two dimensions (x,y) and has a maximum size of 65535 in each dimension. CUDA allows the user to control the arrangement of threads into blocks and of blocks into the grid. The developer can control how these threads are arranged in order to make optimal use of the on-chip memory.

When a CUDA application is executed, each block is assigned to execute on one multiprocessor. This SM will manage and execute the threads within the block on its SPs. While the block arrangement does not have a significant impact on the execution of the threads on the multiprocessors, it strongly affects the manner in which threads access memory and use the memory optimisations available on the GPU. Threads should be arranged into blocks such that the maximum amount of data can be shared between threads in fast on-chip memory and that any unavoidable global memory accesses be coalesced.

4. Generating Complex Networks

Complex networks arising from generator algorithms like the α model have non-trivial topological features, like those often found in real-world networks such as social networks [25], [11], metabolic networks [26], [27] or neural networks [28]. The connections between nodes in a complex network are neither purely regular nor purely random. Many of these networks possess the properties of high clustering and short mean vertex-vertex path length characteristic to small-world networks [9], which explains the interest in this category of networks over the last decade.

As can be seen from the algorithm in Section 2 above, there are some compute-intensive stages involved in generating α -model synthetic networks. The $\mathcal{O}(n^2k)$ complexity of the algorithm arises from the need to compute a property over all vertices for each edge of each vertex. It is therefore attractive to find a way of accelerating or parallelising the algorithm to allow feasible simulation of networks with large n and k .

We have implemented the α -model for and compare the performance between: a single CPU core (Section 4.1), a multi-core CPU utilising all available cores (Section 4.2) and a graphics device used as compute accelerator (Section 4.3).

4.1 The Sequential CPU Implementation

The sequential CPU implementation is used as reference to measure the scaling behaviour of the multi-threaded implementation and to explain the basic steps of the algorithm. Algorithm 1 uses pseudo-code to describe the steps that are executed when generating a graph using the α -model.

Algorithm 1 Pseudo-code for the sequential CPU implementation of Watts' α -model network generator.

```

//generate  $M \leftarrow kn/2$  edges
for  $e \leftarrow 1$  to  $M$  do
  //R is the set of vertices not yet chosen in this round
  if  $R = \{\}$  then
     $R \leftarrow$  init. remaining vertices to set of all vertices  $V$ 
   $v_i \leftarrow$  randomly chosen vertex from  $R$ 
  remove  $v_i$  from  $R$ 
  for all  $v_j \in V, j \neq i$  do
     $p \leftarrow p_{base} \times$  uniform random number
    determine if edge  $e_{i,j}$  exists
    count the neighbours shared by  $v_i$  and  $v_j$ 
     $P[j] \leftarrow$  compute  $v_i$ 's propensity to connect to  $v_j$ 
  normalise the results in  $P$ 
  //select  $v_i$ 's new neighbour  $v_j$ 
   $r \leftarrow$  uniform random number
   $p_{sum} \leftarrow 0.0$ 
  for all  $v_j \in V, j \neq i; p_{sum} < r$  do
     $p_{sum} \leftarrow p_{sum} + P[j]$ 
  insert  $v_j$  into adjacency-list  $A_i$  of  $v_i$ 
  insert  $v_i$  into adjacency-list  $A_j$  of  $v_j$ 

```

The propensity $R_{i,j}$ of vertex v_i to connect to vertex $v_j \in V, j \neq i$ is calculated according to Equation 1. The baseline, random probability of an edge existing is $p_{base} = \binom{n}{2}^{-1}$. To account for the $p \ll p_{base}$ in the equation, the probability p is calculated for every possible edge $e_{i,j}$ by multiplying the baseline probability with a uniform random number.

All implementations of the generator described here insert a new neighbour into an adjacency-list so that the list is sorted in ascending order. This significantly improves performance when counting the neighbours shared by two vertices.

4.2 The Multi-Threaded CPU Implementation

While CPU manufacturers have traditionally increased the CPU frequencies from one generation to the next, this trend has slowed down dramatically over the last years, as the increasing power consumption becomes more and more difficult to manage. Manufacturers like Intel or AMD have instead started to incorporate more cores onto a single CPU die.

The consequence for software developers and end users is that sequential software does not automatically run faster on newer CPUs. Programmers have to re-think and modify their software to utilise multiple threads that can run concurrently on different CPU cores. However, multi-threaded programming is more difficult than writing a sequential implementation. To ease the burden placed on the developers and to increase their productivity, libraries have emerged that attempt to abstract some of the difficulties of multi-threaded programming away from the developer. One of these libraries is the Threading Building Blocks (TBB) library [29] developed by Intel. We have shown previously [30] that it achieves comparable performance to low-level multi-threaded implementations based on POSIX threads [31].

We have used TBB to parallelise the CPU implementation of the α -model. To achieve good performance, the inner-loops need to be parallelised where possible. The following loops described in Algorithm 1 can be executed in parallel:

- The set R of vertices not yet chosen in the current round is initialised using `tbb::parallel_for`.
- The random numbers needed for $p \leftarrow p_{base} \times$ uniform random number are generated using T instances of `tbb::tbb_thread`, where T is the number of logical CPU cores available. Each thread generates n/T random numbers using its own random number generator (RNG) instance to ensure repeatability of the algorithm. The

Algorithm 2 Propensity computation using TBB's `tbb::parallel_reduce`. This code uses the lambda expressions introduced in the upcoming C++0x standard. Lambda expressions let the compiler do the work of creating the function objects needed by TBB, which makes the code easier to read. Each task iterates over a range of values, computing the respective propensity values.

```
double totalPropensity = tbb::parallel_reduce(
    tbb::blocked_range<int>(0, nVertices), 0.0,
    [=](const tbb::blocked_range<int>& range,
        double init)->double {
        //this lambda function computes the
        //propensities for a range of vertices
        double sum = init; //sum over given range
        const int end = range.end();
        for (int i = range.begin(); i < end;++i){
            // compute propensity[i] here
            sum += propensity[i]; // add to sum
        }
        return sum;
    },
    [](double x, double y)->double {
        //lambda function combines two results
        return x+y;
    }
);
```

random numbers are stored in an array for consumption in the next step. Repeatability is also the reason why the random numbers are not generated as part of the following `tbb::parallel_reduce` step. The TBB scheduler assigns the tasks used during the reduction operation dynamically to hardware threads and does not expose this information to the developer, making it difficult to generate the same sequence of random numbers over multiple runs initialised with the same seed.

- The most time consuming loop by far is the computation of v_i 's propensity to connect to all $v_j \in V$. It is however fairly straight forward to parallelise, as there are no dependencies between the iterations except for the calculation of the total propensity sum, which is later on needed to calculate the uniform propensity values. This can be implemented using the `tbb::parallel_reduce` operation as shown in Algorithm 2.

The last remaining inner-loop iterates over the propensity values, normalises these values and computes the inclusive prefix-sum until a random number exceeds the sum, at which point it has found the new neighbour v_j and can stop running. This loop is computationally cheap, runs for only $n/2$ iterations on average, would require a parallel scan over all n elements and another parallel operation to determine v_j and is therefore not worth parallelising on the CPU.

The two function calls needed to create the new edge by inserting the two vertices into each others adjacency-lists (v_j into A_i and v_i into A_j) can be executed in parallel using `tbb::parallel_invoke`. However, this decreased performance slightly during our tests and is thus not done in parallel.

4.3 The CUDA GPU Implementation

Graphical processing units have become popular as compute accelerators for non-graphical applications in the high-performance computing sector over the last years. NVIDIA has helped this trend with their popular CUDA toolkit, which

Algorithm 3 Pseudo-code for the CUDA implementation of Watts' α -model network generator. This is the host code that manages the CUDA execution.

```
allocate device memory incl. sufficient heap memory
generate seeds for the device RNGs and copy to device
do in parallel on the device using T threads: initialise device RNGs
V_deg ← 0 //init. the vertex degree device array
create CUDPP plan for parallel scan on the device
//generate M ← kn/2 edges
for e ← 1 to M do
    //R is the set of vertices not yet chosen in this round
    if R = {} then
        R ← init. remaining vertices to set of all vertices V
        v_i ← randomly chosen vertex from R
        remove v_i from R
    do in parallel on the device using T threads:
        generate n random numbers
    do in parallel on the device using n threads:
        call compute_propensity_kernel
    do in parallel on the device using n threads:
        call normalise_propensity_kernel
    do in parallel on the device:
        call cudppScan (inclusive prefix-sum)
    do in parallel on the device using n threads:
        call select_nbr_kernel
    do in parallel on the device using 2 thread blocks:
        call add_arc_kernel
destroy CUDPP plan and free device memory
```

introduces a few extensions to C++ that enable developers to tap into the processing power of the GPU using well established C programming tools and skills. The GPU code is written in form of so called *device kernels* – routines that execute on the GPU – which are managed by the CPU running the host program. The CPU is free to perform other tasks while the GPU is processing a kernel.

The GPU has a highly data-parallel architecture with many processing units, making it very powerful when executing the same instructions on large arrays of data, but making it difficult to achieve good performance when running algorithms that are more serial in nature, have many conditional branches or are bandwidth-limited. However, we have shown [32], [30], [33], [21] that it is often possible to achieve good speed-ups over traditional CPUs even in such less-than-optimal situations.

The task of generating a graph like the α -model is particularly challenging, as it is necessary to repeatedly iterate over the neighbours structure, which puts high demands on the memory bandwidth and does not perform many compute instructions per data element. But the recent interest in the Graph500 and the newly gained ability to dynamically allocate and free memory in device code makes it an interesting challenge. Algorithm 3 describes the host code that coordinates the device kernel execution.

We use Marsaglia's random number generator [34] to generate both the host and device random numbers. The CUDA implementation of the RNG is explained in [35]. For large networks it would not be feasible to have a separate RNG instance for every vertex, as each instance requires 400 bytes of global memory. We therefore use a specialised RNG kernel, which is executed by T CUDA threads, to generate the random numbers. Every thread generates x random numbers per kernel call, where $x = \text{ceil}(n/T)$. The value of T depends on the graphics hardware used. It must be large enough to keep the device busy and it should be divisible by the number of cores available on the GPU, the number of multiprocessors and the thread block size. We set $T = 30720$ for the GTX580 used for the performance measurements reported in this paper.

Algorithm 4 The `compute_propensity_kernel`.

```

Input parameters:  $v_i, p_{base}, \alpha, k$ 
 $v_j \leftarrow$  global thread ID queried from CUDA runtime
 $sum_s \leftarrow 0.0$  //init. block local propensity sum
 $p \leftarrow p_{base} \times$  (random uniform number)
 $k_i \leftarrow$  load degree of vertex  $v_i$ 
 $A_i(ptr) \leftarrow$  load the pointer to adjacency-list  $A_i$ 
 $A_j \leftarrow$  load adjacency-list at  $A_i(ptr)$  into shared memory
synchronise thread block
determine if edge  $e_{i,j}$  exists ( $v_j \in A_i$ )
if edge does not exist then
   $k_j \leftarrow$  load degree of vertex  $v_j$ 
   $A_j(ptr) \leftarrow$  load the pointer to adjacency-list  $A_j$ 
  count the neighbours shared by  $v_i$  and  $v_j$ 
   $p \leftarrow$  compute  $v_i$ 's propensity to connect to  $v_j$ 
 $sum_s[btid] \leftarrow p$  //btid is the thread idx within the block
 $P[v_j] \leftarrow p$  //write propensity to global memory
synchronise thread block
 $sum_s \leftarrow$  perform reduction operation
// $sum_s[0]$  now contains the block local sum
if  $btid = 0$  then
  atomically add the local sum  $sum_s[0]$  to the global sum

```

Algorithm 5 Device function `countSharedNbrs` counts the number of neighbours shared by two vertices. It assumes that the adjacency-lists are sorted in ascending order. The performance of this function is critical to the overall performance. A_i is stored in shared memory.

```

__device__ __forceinline__
Uint countSharedNbrs(int v_i, int* A_i,
                    int v_j, int* A_j) {
  Uint count = 0;
  for (int idx1=0, idx2=0, tmp;
       idx1 < v_i && idx2 < v_j;) {
    count = A_i[idx1] == A_j[idx2] ? count + 1 : count;
    tmp = A_i[idx1] < A_j[idx2] ? idx1 + 1 : idx1;
    idx2 = A_j[idx2] < A_i[idx1] ? idx2 + 1 : idx2;
    idx1 = tmp;
  }
  return count;
}

```

Algorithm 4 shows how the propensity values are computed on the device. A_i can be loaded to shared memory using coalesced memory transactions as all threads in the block cooperate to load it from global memory. The data in A_i is used by all threads. A_j is different for every vertex and the global memory transactions are not coalesced, but the automatic caching on Fermi devices helps to keep the throughput relatively high. Algorithm 5 shows the device function that is called to count the shared neighbours. It has to loop over both adjacency-lists and is therefore highly critical to the performance.

After the propensity values and propensity sum have been computed, they need to be normalised. This is all that `normalise_propensity_kernel` needs to do. The array of normalised propensity values is then passed to

Algorithm 6 The `select_nbr_kernel`.

```

Input parameters: random number  $rnd$ 
 $v_j \leftarrow$  global thread ID queried from CUDA runtime
 $P_s[btid + 1] \leftarrow$  load scan results from  $P[v_j]$  into shared mem.
if  $btid = 0$  then
  if  $v_j > 0$  then
     $P_s[0] \leftarrow$  load last value from previous block  $P[v_j - 1]$ 
  else
     $P_s[0] \leftarrow 0$  //first subinterval, no lower value
//determine if  $rnd$  falls into the subinterval  $P_{i,j}$ 
if  $P_s[btid] \leq rnd$  AND  $P_s[btid + 1] > rnd$  then
  write  $v_j$  to mapped host memory

```

Algorithm 7 The `add_arc_kernel`.

```

Input parameters:  $v_i, v_j$ 
 $v \leftarrow v_i$  for block 0 and  $v_j$  for block 1
 $nbr \leftarrow v_j$  for block 0 and  $v_i$  for block 1
 $k_v \leftarrow$  load degree of vertex  $v$ 
if  $btid = 0$  AND  $k_v > 0$  then
   $A_{v,old}(ptr) \leftarrow$  load the pointer to  $A_v$  into shared mem.
if  $btid = 0$  AND need to allocate new  $A_v$  then
   $A_v(ptr) \leftarrow$  allocate memory for  $k_v + 32$  integers
else if  $btid = 0$  then
   $A_v(ptr) \leftarrow A_{v,old}(ptr)$  //keep using the existing array
synchronise thread block
 $A_v \leftarrow$  load adjacency-list at  $A_{v,old}(ptr)$  into shared mem.
copy elements from  $A_v$  to  $A_v(ptr)$ , adding one to the index if the current value is larger than  $nbr$ 
if  $btid = 0$  then
  insert  $nbr$  into  $A_v(ptr)$  so that  $A_v(ptr)$  is sorted (asc.)
  write  $k_v + 1$  to global memory
if  $A_v(ptr) \neq A_{v,old}(ptr)$  then
  write the pointer  $A_v(ptr)$  to global memory
  if  $k_v > 0$  then
    free memory pointed to by  $A_{v,old}(ptr)$ 

```

`cudaPpScan`, a function of the CUDA Data Parallel Primitives Library [36] (CUDPP), which performs an inclusive parallel prefix-sum operation on the input data.

The result of the scan operation is the array $P_{i,j}, \forall j \neq i$ of subintervals in the range $[0,1)$. This is passed to `select_nbr_kernel`, along with a uniform random number. The kernel, described in Algorithm 6, then determines for every v_j if the random number falls into the respective subinterval. To do this, each thread checks whether the random number is larger than or equal to the upper end of the propensity range for the previous vertex $P[v_j - 1]$ and smaller than the upper end of the propensity range $P[v_j]$. This is the case for exactly one vertex v_j , which is selected as the new neighbour.

Finally, the edge $e_{i,j}$ can be created. This is done in Algorithm 7 `add_arc_kernel`. This kernel is only executed by 2 thread blocks, one for each end of the new edge, which it inserts into the respective adjacency-list. This only utilises a fraction of the processing units on the device (16 multiprocessors with a total of 512 processing units on the GTX580). To improve device utilisation, we use two CUDA streams to concurrently run the `add_arc_kernel` and generate the random numbers needed for the next iteration. These two kernels have no dependencies on each other and can therefore safely run in parallel. The `add_arc_kernel` does not allocate new memory every time it adds a new vertex. Instead, it increases the allocated length by 32 elements when needed. This is critical to performance, as the malloc operations are expensive. The minimum increment should be 4 for 32-bit arrays, as the pointers returned by malloc are aligned to 16 bytes. However, 32 works nicely with the 128 byte cache line width on Fermi devices and performed better in our experiments. The downside of allocating too much is wasted memory.

We will refer to this CUDA implementation as CUDA 1. We have also implemented a second version of the algorithm, which we shall refer to as CUDA 2 respectively, that is identical to the first implementation except for one key criterion: It replaces the `add_arc_kernel` with a kernel that merely determines the indices into the adjacency-lists at which the new neighbours have to be inserted to keep them sorted. The indices are written to mapped host memory. The actual memory allocation, deallocation and copying is initiated in the traditional way by the host thread with calls to `cudaMalloc`, `cudaFree` and `cudaMemcpy(Async)`.

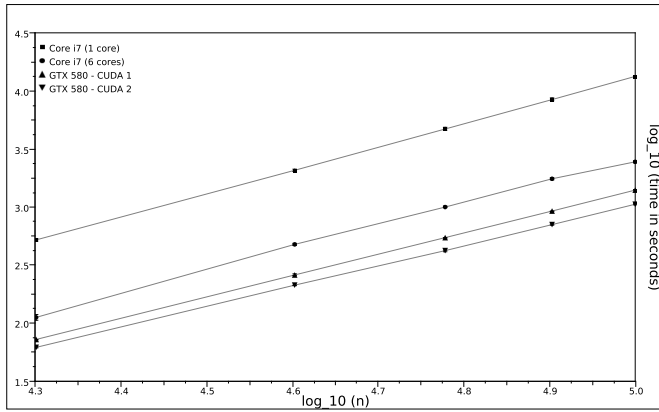


Fig. 2: This plot shows how the algorithms scale with the network size $n = 20\,000$ to $100\,000$. The mean degree is set to $k = 20$ and $\alpha = 1.0$. The standard deviations showing the measurement error are smaller than the symbol size.

This second implementation is used to compare the performance of the new in-kernel functions for dynamic memory management to the library calls initiated by the host thread.

5. Performance Results

To compare the performance of the different implementations of the α -model, we measured the execution time for each of the algorithms. We tested the scaling behaviour with increasing network size and fixed degree and vice versa. The parameter α is always set to 1.0 for these experiments. Other values of α only have a very minor impact on the performance.

The test platforms run Ubuntu Linux 10.10 with GCC 4.5 and CUDA 3.2. Both the single-threaded and multi-threaded CPU implementations were tested on an Intel Core i7 970 with 6 cores running at 3.2 GHz. This machine has 12 Gbytes of main memory. The CUDA algorithm was tested on an Intel Core i7 870 with 4 cores running at 2.93 GHz, 4 Gbytes of main memory and an NVIDIA GTX 580 GPU.

Figure 2 compares the performance with increasing network size and fixed degree $k = 20$. For the largest measured network size $n = 100\,000$, the multi-core TBB implementation runs 5.42 times faster than the sequential algorithm and thus scales well with the number of cores. The GPU implementation CUDA 1 runs 1.75 times faster than the TBB implementation and 9.50 times faster than the sequential CPU code. Implementation CUDA 2 is 2.33 times and 12.64 times faster than the sequential and TBB CPU implementations respectively.

Figure 3 compares the performance with increasing degree and fixed network size $n = 20\,000$. For the largest degree $k = 100$, the multi-core TBB implementation runs 7.16 times faster than the sequential implementation. Intel's Hyper-Threading Technology allows each physical processor core to appear as two logical cores and to work on two tasks at the same time, improving the utilisation of the physical core and increasing throughput. The speed-up value shows that this technology can pay-off, allowing the algorithm to achieve performance values higher than possible by the increase of physical cores alone. At this degree, the TBB implementation even slightly outperforms implementation CUDA 1 running on the graphics device by 1.03 times. The small kinks in the performance scaling of CUDA 1 are caused by the

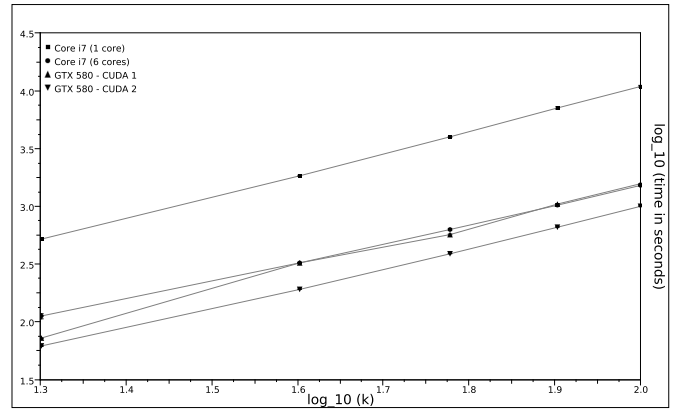


Fig. 3: This plot shows how the algorithms scale with the average degree $k = 20$ to 100 . The network size is set to $n = 20\,000$ and $\alpha = 1.0$. The standard deviations showing the measurement error are smaller than the symbol size.

fixed increment size of 32 elements for the adjacency-list lengths. The host library calls for memory management have a much smaller impact on the overall performance, and for that reason the results for CUDA 2 do not show the same anomalies. CUDA 2 is 1.51 times faster than the TBB implementation and 10.83 times faster than the sequential CPU implementation.

These speed-ups are not as impressive as those seen for more regular or compute heavy algorithms, but the results show that the GPU can still consistently outperform the CPU even for a graph generator problem like the one described here. The new device code `malloc()` and `free()` functions, while arguably more elegant, have been shown to be significantly more expensive than the proven method of handing device memory management tasks to the host.

6. Discussion

The addition of new features like dynamic memory allocation in device code makes CUDA an increasingly powerful environment that enables programmers to tap into the parallel processing power of today's graphics processing units. And while GPUs have received lots of attention for their impressive number crunching capabilities, which have enabled GPU based supercomputers to take up the top spots in the TOP500 benchmark, they are not always seen as the best choice when it comes to memory bound or irregular problems. However, such algorithms are essential for a wide variety of applications, and the introduction of the Graph500 highlights the importance of graph based algorithms for high performance computing.

While it is true that the speed-ups achieved for many graph based algorithms are not as impressive as those for compute bound algorithms, we have shown time and time again that they can still be very respectable if the code and data structures are optimised for the GPU hardware. The graph generation problem discussed in this article again proves this point. As shown in Table 1, the GPU manages to consistently outperform a high-end six core Intel CPU of the same vintage when managing the device memory from the host (CUDA 2) and at least keeps up with the multi-threaded CPU implementation when using in-kernel dynamic memory management (CUDA 1). A high connectivity in the α -model algorithm is particularly taxing for the graphics

Table 1: Summary of the performance results. The speed-up values are relative to the respective sequential CPU implementation and the quoted slopes are for the least squares linear fits to the data sets.

Compute Device	Speed-up	Slope
n=100,000 ; k=20 ; $\alpha=1.0$		
Core i7 970 (1 core)	1.00	2.02
Core i7 970 (6 cores)	5.42	1.89
GTX 580 - CUDA 1	9.50	1.84
GTX 580 - CUDA 2	12.64	1.76
n=20,000 ; k=100 ; $\alpha=1.0$		
Core i7 970 (1 core)	1.00	1.90
Core i7 970 (6 cores)	7.16	1.74
GTX 580 - CUDA 1	6.96	1.87
GTX 580 - CUDA 2	10.83	1.76

hardware, as it makes it necessary to repeatedly iterate over ever growing adjacency-lists, further increasing the demands on the memory interface.

7. Conclusions

We have shown how Watts' α -model of small-world graphs can be used to generate synthetic network realisations on multi-core CPUs and with GPU accelerators. We deliberately chose a hard "pointer-chasing" problem to see how a GPU implementation might cope, given the memory system constraints of the GPU architecture model. The more commonly studied Watts-Strogatz β -model [20] is generally easier to compute and presents less need for a parallel generation than the α -model.

As discussed GPUs are a highly topical and popular subject for acceleration of calculations and we expect this trend to continue. There is plenty of work reported in the literature – including some by ourselves – on how GPUs can routinely provide speed-up factors of over one hundred for regular geometric problems. It is therefore interesting to observe that even for highly irregular and unbalanced problems, such as generating α -model networks, GPUs still provide some performance utility as accelerators to the CPU. Given the cost-performance properties of a GPU compared with a typical CPU we judge the performance speed-ups reported for the GPU well worthwhile.

We also note the sensitivity of choice of GPU memory mapping used for the algorithmic work reported here. The CUDA programming model is a powerful one, recent additions such as the in-kernel malloc capability offer additional power but there is still scope for building up GPU memory mapping experience for GPU/CUDA implementations of graph problems such as we have described. As so often with CUDA, there is a trade-off between flexibility and performance when it comes to memory management. The flexibility gained by the device code memory management functions comes at a cost compared to device memory management initiated by the host thread.

There is scope for also parallelising some of the graph network analysis algorithms such as the path-length and clustering coefficient metrics. Implementing these to work with data already in GPU memory offers some further speed-up potential for statistical analysis of synthetic networks. We also anticipate a need to investigate hybrid synthetic networks where a large network may comprise a combination of models such as the α - and β - small-world models and scale-free and other structures.

References

- [1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.
- [2] R. W. Floyd, "Algorithm 97: Shortest Path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.
- [3] T. Schank and D. Wagner, "Approximating clustering coefficient and transitivity," *Journal of Graph Algs. & Apps.*, vol. 9, no. 2, pp. 265–275, 2005.
- [4] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, March 1975.
- [5] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, p. 169, 2003.
- [6] A.-L. Barabasi, *Linked - The New Science of Networks*. Perseus, 2002.
- [7] P. Erdős and A. Rényi, "On random graphs," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.
- [8] E. N. Gilbert, "Random Graphs," *Annals of Mathematical Statistics*, vol. 30, no. 4, pp. 1141–1144, 1959.
- [9] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, June 1998.
- [10] A.-L. Barabasi and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, October 1999.
- [11] M. E. J. Newman, "The structure of scientific collaboration networks," *PNAS*, vol. 98, no. 2, pp. 404–409, January 2001.
- [12] —, "Finding community structure in networks using the eigenvectors of matrices," *Phys. Rev. E*, vol. 74, pp. 036104–1–19, 2006.
- [13] D. Liben-Nowell and J. Kleinberg, "Tracing information flow on a global scale using Internet chain-letter data," *PNAS*, vol. 105, no. 12, pp. 4633–4638, March 2008.
- [14] M. E. J. Newman, S. H. Strogatz, and D. J. Watts, "Random graphs with arbitrary degree distribution and their applications," *Phys. Rev. E*, vol. 64, no. 026118, 2001.
- [15] J. Kleinberg, "The small-world phenomenon: an algorithmic perspective," in *Proc. of the 32nd annual ACM symp. on Theory of comp.*, 2000.
- [16] R. Albert and A. Barabasi, "Statistical mechanics of complex networks," *Rev. Mod. Phys.*, vol. 74, no. 1, pp. 47–97, January 2002.
- [17] Graph500.org, "The Graph 500 List," <http://www.graph500.org/>.
- [18] TOP500.org, "TOP 500 Supercomputer Sites," <http://www.top500.org/>.
- [19] NVIDIA CUDA™ C Programming Guide Version 3.2, NVIDIA® Corporation, 2010. [Online]. Available: <http://www.nvidia.com/>
- [20] D. J. Watts, *Small worlds: the dynamics of networks between order and randomness*. Princeton University Press, 1999.
- [21] A. Leist, D. Playne, and K. Hawick, "Exploiting Graphical Processing Units for Data-Parallel Scientific Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, pp. 2400–2437, December 2009.
- [22] I. Buck, T. Foley, D. Horn, J. Sugeran, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004.
- [23] *Technical Overview ATI Stream Computing*, ATI, 2009. [Online]. Available: http://developer.amd.com/gpu.assets/Stream-Computing_Overview.pdf
- [24] M. McCool and S. D. Toit, *Metaprogramming GPUs with Sh*. A K Peters, Ltd., 2004.
- [25] S. Milgram, "The Small-World Problem," *Psychology Today*, vol. 1, pp. 61–67, 1967.
- [26] H. Jeong, B. Tombor, R. Albert, Z. Oltvai, and A.-L. Barabasi, "The large-scale organization of metabolic networks," *Nature*, vol. 407, no. 6804, pp. 651–654, October 2000.
- [27] D. A. Fell and A. Wagner, "The small world of metabolism," *Nature Biotechnology*, vol. 18, no. 11, pp. 1121–1122, November 2000.
- [28] D. S. Bassett and E. Bullmore, "Small-world brain networks," *The Neuroscientist*, vol. 12, pp. 512–523, 2006.
- [29] Intel(R), *Threading Building Blocks Reference Manual*, Intel, May 2010.
- [30] K. Hawick, A. Leist, and D. P. Playne, "Mixing Multi-Core CPUs and GPUs for Scientific Simulation Software," *Res. Lett. Inf. Math. Sci.*, vol. 14, no. ISSN 1175-2777, pp. 25–77, 2010. [Online]. Available: <http://www.massey.ac.nz/massey/learning/departments/iims/research/research-letters/>
- [31] IEEE, *IEEE Std. 1003.1c-1995 thread extensions*, 1995.
- [32] K. Hawick, A. Leist, and D. Playne, "Regular Lattice and Small-World Spin Model Simulations using CUDA and GPUs," *Int. J. Parallel Prog.*, vol. 39, no. 2, pp. 183–201, 2011.
- [33] K. A. Hawick, A. Leist, and D. P. Playne, "Parallel Graph Component Labelling with GPUs and CUDA," *Parallel Computing*, vol. 36, pp. 655–678, 2010. [Online]. Available: www.elsevier.com/locate/parco
- [34] G. Marsaglia, A. Zaman, and W. W. Tsang, "Toward a universal random number generator," *Statistics and Probability Letters*, vol. 9, no. 1, pp. 35–39, January 1987, florida State preprint.
- [35] K. Hawick, A. Leist, D. Playne, and M. Johnson, "Speed and Portability issues for Random Number Generation on Graphical Processing Units with CUDA and other Processing Accelerators," in *Proc. Australasian Computer Science Conference (ACSC 2011)*, 2011.
- [36] M. Harris, J. D. Owens, S. Sengupta, S. Tzeng, Y. Zhang, and A. Davidson, *CUDPP: CUDA Data Parallel Primitives Library*, accessed Jan. 2011.

Hierarchical Parallelization of Molecular Fragment Analysis on Multicore Cluster

Liu Peng¹, Bhupesh Bansal¹, Ashish Sharma²,

Rajiv K. Kalia¹, Aiichiro Nakano¹, Priya Vashishta¹

¹Laboratory for Advanced Computing and Simulations

University of Southern California, Los Angeles, CA, 90089, USA

Email: {liupeng, rkalia, anakano, priyav}@usc.edu, bbansal.usc@gmail.com

²Center for Comprehensive Informatics

Emory University, Atlanta, GA 30322, USA

Email: Ashish.Sharma@emory.edu

Abstract—Molecular fragment analysis, using connected component identification algorithm, is of great significance for structural and chemical analysis in computer aided material design. However, it is a great challenge to accelerate molecular fragment analysis due to the scale, diversity and irregularity of molecular graphs. To address this challenge, we propose a hierarchical parallelization approach consisting of: (1) inter-node parallelization via spatial decomposition and hook-and-contract algorithm; (2) inter-core parallelization via master-and-worker scheme; and (3) locality optimization based on space-filling curve to improve memory accessing. Experiments show that the proposed scheme achieves nearly linear inter-node strong scalability up to 50 million vertices molecular graph on 32 computing nodes, and over 13-fold inter-core speedup on 16 cores. The experiments also demonstrate the effectiveness of locality optimization on performance enhancement.

1. Introduction

Massive data analysis on parallel computers has become an essential part, and often a bottleneck, of scientific computing. For example, large-scale molecular dynamics (MD) simulation, which has become an integral part of computer aided material design, often involves multimillion atoms and computing the molecular fragments (or connected components) is essential for the structural and chemical analysis, such as identifying molecular products during the combustion of fuels [1]. Challenges of molecular fragment analysis mainly arise from three aspects: (1) complexity introduced by scale of molecular graph—high-end MD simulations typically involve multimillion atoms [2], which amounts to multimillion vertex-sized graph, imposing great difficulty for molecular fragment identification, and therefore it is important to design efficient parallel algorithm to harvest computing power; (2) diversity introduced by molecular graph density and connectivity—due to the wide range of MD simulations, for example, simulation sizes can range from

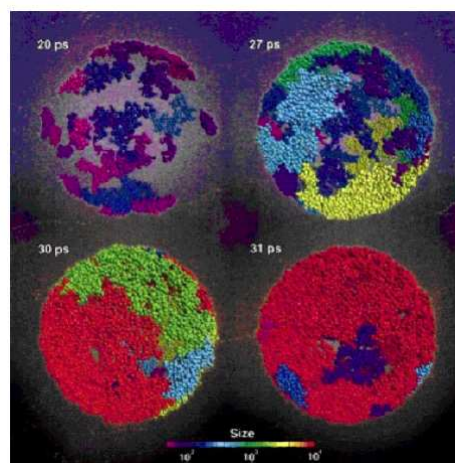


Fig. 1: Snapshot of a molecular dynamics simulation to study the percolation of OAl_4 cluster during the oxidation of an aluminum nanocluster, where color represents the size of molecule fragments.

100,000 atoms (10MB of data per frame) to 1 billion atoms (100GB), the resulting molecular graph can vary from dense to sparse, from a small number of large fragments to a large number of small fragments. Fig. 1 illustrates the variation in data characteristics starting from a large number of small molecular fragments at the beginning of the simulation to a nearly connected single fragment in the last snapshot with the increased percolation level; (3) irregularity introduced by MD datasets—MD datasets are inherently irregular, so are the molecular graph generated from them, and serial access to the molecular graph datasets often exhibits poor spatial and temporal locality, which leads to ineffective use of memory hierarchy [3]. Therefore, it is a great challenge to accelerate molecular fragment analysis.

Problem Statement: Let $G = (V, E)$ represent an undirected

molecular graph, where the vertex set V consists of atoms and the edge set E contains bonds between pairs of atoms. Let i and j represent two atoms; then edge $(i, j) \in E$ iff there is a bond between atom i and atom j . The problem is to identify all connected components in the molecular graph. We refer to vertex as an atom in the graph.

There are some special features in the molecular graph:

- *Vertex degree restriction:* The vertex degree has an upper limit. For example, in the simulation of SiO_2 , the vertex degree, for each silicon atom, is typically four and, for each oxygen atom, is on average two. Due to the nature of the chemical bonding, even outliers do not exceed the maximum node degree, typically set around 10. And this restricts the number of expanding vertices in the breadth first search (BFS) algorithm.
- *Boundary restriction:* Due to the active interatomic interaction cutoff R_c , scientists are usually interested in only atoms within the cutoff distance. This restricts the depth in the depth first search (DFS) algorithm.

The major contributions of this paper is a hierarchical parallelization approach on a multicore cluster, combining:

- Spatial decomposition and hook-and-contract algorithm are utilized effectively for inter-node parallelization.
- Master-and-worker scheme is used for inter-core parallelization.
- Locality optimization via space-filling curve is employed to improve memory accessing.

The rest of this paper is organized as follows. After summarizing related works in section 2, section 3 presents the hierarchical parallelization scheme, including hierarchical decomposition, inter-node and inter-core parallelization. And section 4 details our locality optimization via space-filling curve. Section 5 presents the experimental results and analysis. Finally, section 6 concludes the paper.

2. Related Work

There have been great efforts on the study of connected component identification. Generally, there are three approaches : (1) Graph traversal, such as DFS and BFS based approach; (2) Graph adjacency matrix transitive closure based approach; (3) Hook-and-contract based approach, where vertices are hooked together to form a large set of vertices and then outgoing duplicate edges and internal edges to the set are removed to contract the set to a single super-vertex, and the process is repeated until the maximally connected components are found. Hirschberg, Chandra, and Sarwate [4] presented an $O(\log^2 n)$ algorithm using $n^2/\log n$ processors on the CREW PRAM model, where n is the number of vertices. Chin [5] later reduced the processor requirement to $n^2/\log^2 n$. Johnson and Metaxas [6] proposed a CREW algorithm with $O(\log^{1.5} n)$ complexity using $n+m$ processors, where m is the number of edges. Chong and Lam [7] presented an $O(\log n \log \log n)$ algorithm using

$n + m$ processors for the EREW model. However, these fine-grained parallel algorithms require impractical number of computing nodes for molecular fragment analysis which usually involves millions of atoms. Consequently, researchers have also studied coarse-grained parallel algorithms for connected component analysis. Chin [5] designed an $O((m \log n)/p + \log n)$ algorithm for up to $p = n/\log n^2$ processors, in PRAM model, for dense graphs. Kruskal [8] presented faster parallel algorithms for sparse graph with $O(m/p + (n \log p)/p + p^{1+\epsilon})$ complexity for the EREW model and $O(m/p + (n \log p)/p + p \log p)$ for the CREW model, where ϵ is a negligible value. All these approaches assume shared memory, which is not the case for cluster—the current most popular platform.

3. Hierarchical Parallelization

In this section, we present our hierarchical hook-and-contract algorithm. In the following subsections, we detail inter-node parallelization and inter-core parallelization, respectively.

3.1 Inter-node Parallelization

In this subsection, we first describe how to decompose the molecular graph and assign decomposed subgraphs to computing nodes. Then we describe a hook-and-contract algorithm for inter-node parallelization implemented via Message Passing Interface (MPI).

- *Inter-node decomposition.* 3D mesh decomposition is employed to divide the whole molecular graph into smaller subgraphs, and each subgraph is mapped to a processor in an array of $P_x \times P_y \times P_z$ computing nodes, where P_x , P_y , and P_z are even positive integers. The purpose of decomposition is to assign equal load to each computing node, thereby achieving load balancing, which is known to be an effective way to improve the parallelization efficiency of irregular applications [9]. Specifically, our scheme partitions the whole molecular graph (in terms of atoms) in a computational space, which is related to the physical space by a curvilinear coordinate transformation: The computational space shrinks where the workload density is high and expands where the density is low, so that the workload is uniformly distributed. To minimize the load imbalance and communication costs as a function of the coordinate transformation, our approach employs simulated annealing to figure out the optimal range information. The range information of each computing node is then propagated to its 26 neighbors in a 3D cube.
- *Inter-node hook-and-contract algorithm.* After the spatial decomposition, each computing node is assigned a chunk of atomic data, i.e. a list of a subset of atoms. Then, in step 7 of Alg. 1, each node performs independent *Singlenode_Fragment_Analysis* using a graph traversal algorithm shown in Alg. 2. This

Algorithm 1: Inter-node parallel algorithm for molecular fragment analysis. n is the number of atoms and p is the number of computing nodes.

input : 1. an atom bond file containing adjacency lists of atoms.
2. an atom dataset file containing atom attributes¹.

output: Fragment list

- 1: Each node gets a chunk of input files.
- 2: **for** node 1 to p in parallel **do**
- 3: Load balancing to distribute atoms at each node;
- 4: Each node gets x, y, z range information for its 26 direct neighbors;
- 5: **end for**
- 6: **for** Node 1 to p in parallel **do**
- 7: Run *Singlenode_Fragment_Analysis*;
- 8: **end for**
- 9: **for** i from 1 to $\log_2 p$ **do**
- 10: **for** j from 0 to $(p-1)/2^i$ in parallel **do**
- 11: Node pair $(1 + j \times 2^i, 1 + j \times 2^i + 2^{i-1})$ hook together: (Fragments, adjacency lists and cross-atom lists are merged into node $1 + j \times 2^i$.)
- 12: **end for**
- 13: **end for**
- 14: Return fragments from node 1.

¹ Atom dataset file is used to filter atomic dataset based on some simulation attribute if needed

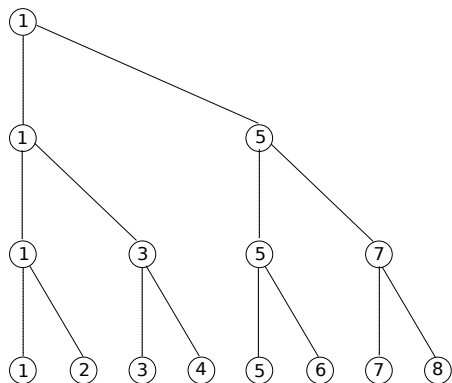


Fig. 2: Inter-node merge example for 8 nodes.

analysis produces three outcomes: (1) it tags each atom with appropriate fragment identification number; (2) it saves local fragments; and (3) it saves the list of cross-atoms connected to each fragment, where a cross-atom is defined as follows: For vertex i, j , if $(i, j) \in E$ then j is a cross-atom of i iff i and j belong to different nodes. This step is done in $O(n'/q + q)$ time where n' and q are the number of atoms and number of cores per computing node respectively.

Algorithm 2: Inter-core parallel algorithm for molecular fragment analysis.

input : Atom list A . $A[i].nbr$ is atom $A[i]$'s adjacency list, $A[i].threadId$ is initialized to -1. t is the number of threads.

output: Local fragment list FL

Master thread:

- 1: Shuffle the atom list A .
- 2: **while** not all atoms in A marked **do**
- 3: Pick the first t unmarked atoms in A and store them in $root$.
- 4: Create a new graph G' with t vertices and no edges.
- 5: $\forall 1 \leq i \leq t$, create fragment $frag_i$.
- 6: Start t worker threads.
- 7: Wait for all worker threads to finish.
- 8: Run BFS to identify connected components in G' .
- 9: Merge fragments inside one connected component in G' .
- 10: Append distinct $frag_i$ to FL .
- 11: **end while**

Worker thread i :

- 1: do a depth-limited search starting from $root[i]$:
- 2: **for** each atom a visited during search **do**
- 3: Put a in $frag_i$ and mark a 's fragment id as i .
- 4: If a 's fragment id is already marked as j , create an edge between vertex $v[i]$ and $v[j]$ in G'
- 5: **end for**

Steps 9-12 in Alg. 1 work as a hook-and-contract mechanism, which consists of $\log_2 p$ iterations. At the i th iteration, $(p-1)/2^i$ group of pair-wise nodes does information merge in parallel. Specifically, nodes $1 + j \times 2^i$ and node $1 + j \times 2^i + 2^{i-1}$, for $j = 0$ to $(p-1)/2^i$, merge their fragment lists, adjacency lists and cross-atom lists to node $1 + j \times 2^i + 2^{i-1}$. Merging fragment list may merge two fragments, if one fragment contains any cross-atom of atoms from the other fragment, into one fragment. After $\log_2 p$ iterations, all merges are done and we get global graph-level cross-node fragments. Fig. 2 illustrates this hook-and-contract scheme. Each tree node represents a computing node in cluster. During iteration 1, group of nodes (1, 2) merge their fragment lists, adjacency lists and cross-atom lists to node 1. Groups (3, 4), (5, 6) and (7, 8) do the same merge to node 3, 5, 7 respectively. After this iteration, nodes 1, 3, 5, 7 together carry all fragments information. During iteration 2, groups (1, 3) and (5, 7) do the same merge again with results in node 1 and 5 respectively. Finally during iteration 3, group (1, 5) merges to node 1, which carries the final fragment information.

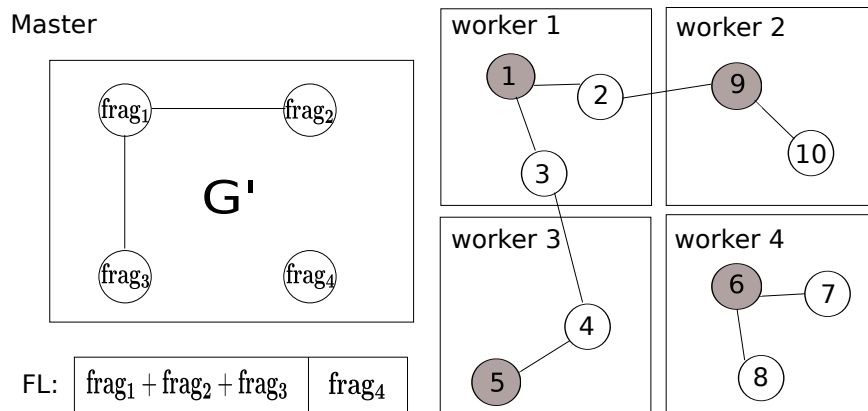


Fig. 3: Master/worker inter-core example.

3.2 Inter-Core Parallelization

Alg. 2 shows the inter-core parallelization for fragment analysis. A master/worker model is employed to accelerate local fragment analysis: Each worker thread independently detects a local fragment through graph traversal algorithm and then the master thread is in charge of merging fragments from each worker thread.

Specifically, the master thread shuffles the whole atom list at step 1. The purpose is to make the t atoms (t is the number of workers), to be used as DFS root by each worker thread, selected at step 3 randomized. At step 4, the master thread creates a new graph G' with vertex i representing the fragment $frag_i$ to be generated by worker thread i . Next, t worker threads are started and they work in parallel to detect $frag_i$ and create edges in G' . After all worker threads are finished, the master thread runs BFS to detect connected components in G' . At step 9, any two fragments belonging to the same connected component in G' are merged. Final fragments are added to local fragment list at step 10.

Each worker thread does a depth-limited DFS based graph traversal to take advantage of material characteristics, e.g. interatomic interaction cutoff distance. At step 3, each thread i keeps marking unvisited atoms as part of $frag_i$. If it encounters an atom already marked in $frag_j$, then it creates an edge between vertex i and j in G' as described at step 4.

Fig. 3 illustrates inter-core parallel algorithm involving one master and four workers. The left part describes the graph G' created by the master, where each vertex $frag_i$ represents the fragment generated by worker i and the edges are created by workers. On the right side, each vertex represents an atom and DFS root atoms are shaded. Each rectangle shows a worker's working region. Workers 1, 2, 3, 4 run depth-limited DFS, in parallel, starting from roots 1, 9, 5, 6, respectively, keeping expanding their working regions independently. When one worker i expands to an atom already visited by another worker j , then an edge is created in G' for $frag_i$ and $frag_j$. For example, when worker 1 tries to expand from atom 2 to

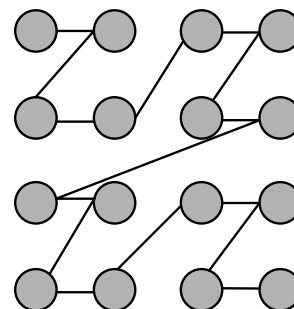


Fig. 4: Z Curve to sort the input data.

atom 9 which is already visited by worker 2, an edge between $frag_1$ and $frag_2$ is added to graph G' . After all workers finish, the master traverses G' , merge $frag_1$, $frag_2$, $frag_3$ into one fragment, denoted as $frag_1 + frag_2 + frag_3$, and append all fragments to FL . The above procedures repeat until all atoms are visited.

4. Locality Optimization

MD datasets are irregular, and serial access to them often exhibits poor spatial and temporal locality, which leads to ineffective use of a memory hierarchy [3]. It has been shown that data reordering (i.e., strategy to change data management to increase data reuse in memory) can significantly improve memory hierarchy utilization [3]. In MD simulations, a space-filling curve (e.g., Morton curve or Z curve illustrated in Fig. 4), is often used to preserve spatial locality [10]. A space-filling curve is a mapping of a one-dimensional array to three-dimensional grid points, which preserves the spatial proximity of successive array elements [3]. Our optimization strategy organizes input atomistic simulation data according to a space-filling curve. We implemented Z curve to improve the data locality and further to enhance the performance of molecular fragment analysis.

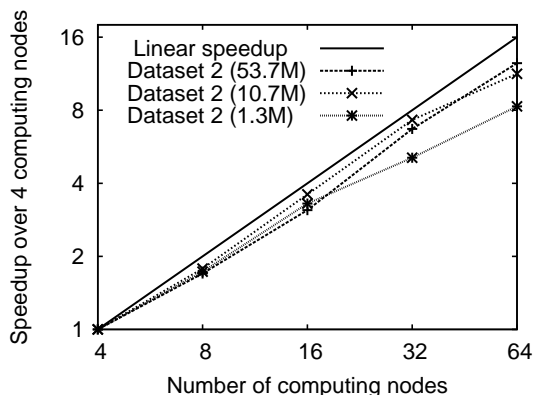


Fig. 5: Inter-node strong-scaling speedup comparison with different sizes of RDX crystal datasets2 up to 64 computing nodes.

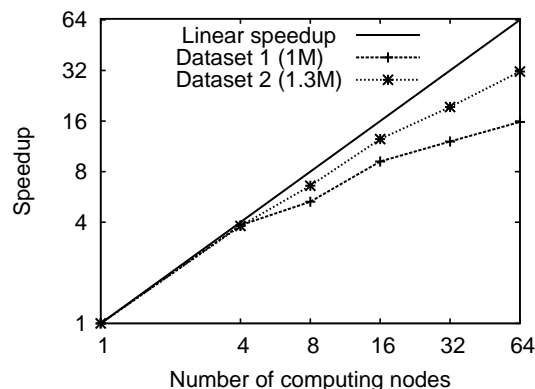


Fig. 6: Inter-node strong-scaling speedup comparison for 1 million-atom shocked RDX dataset1 and 1.3 million-atom RDX crystal dataset2 up to 64 computing nodes.

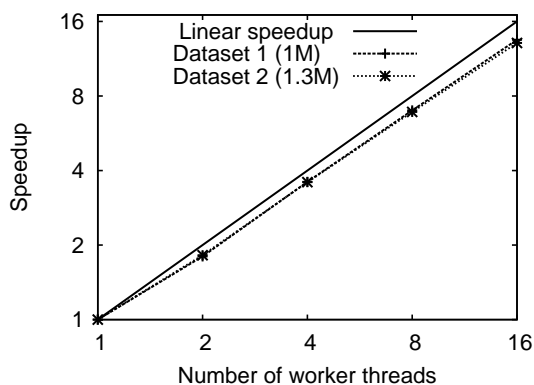


Fig. 7: Inter-core strong-scaling speedup comparison for 1 million-atom shocked RDX dataset1 and 1.3 million-atom RDX crystal dataset2 up to 16 worker threads.

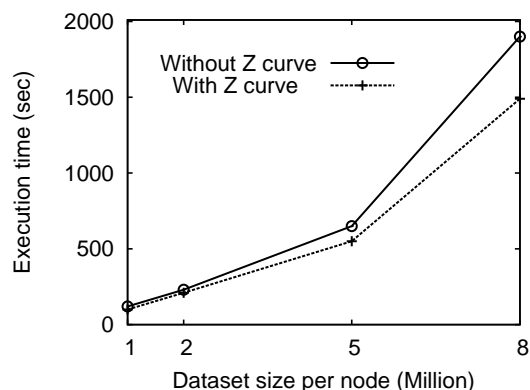


Fig. 8: Execution time as a function of dataset size per node in million atoms with and without Z curve technique.

5. Performance Evaluation

To test the effectiveness of the proposed optimization methods, we conduct three sets of experiment testing the performance of inter-node parallelization, inter-core parallelization and locality optimization.

First, we test the performance of inter-node parallelization. Strong scalability is used as a metric for fixed-size problem. We have conducted scalability tests with two datasets, each of different sizes—approximately 1 million atoms, 10 million atoms and 50 million atoms—on a Linux cluster consisting of 133 Intel Xeon dual-processor (2.8GHz) nodes with 2GB memory per node and connected by Myrinet interconnect. Dataset 1 is an atomistic dataset for shock-compressed RDX (1,3,5-trinitro-1,3,5-triazine) crystal with a large number of long chains and highly dense molecules, whereas dataset 2 represents a normal-density RDX crystalline structure with regular molecules having small chains of atoms and moderate density.

Fig. 5 compares the strong-scaling speedup over 4 computing nodes for different sizes—1.3, 10.7 and 53.7 million atoms—of the RDX crystal dataset (dataset 2). The speedup is nearly linear up to 32 computing nodes for 10.7 and 53.7 million atoms, while it decreases to less than 32 for 64 computing nodes. The reason is that as the number of computing nodes increases, the hierarchical merge takes more time, which degrades the performance. Fig. 6 compares the speedup of two datasets of a similar size but different characteristics (1 million-atom shocked RDX data, dataset 1, and 1.3 million-atom RDX crystal, dataset 2). Dataset 2 exhibits better speedup, which is probably due to the smaller number of cross-atoms.

Next, we test the inter-core strong scalability. We use the same datasets as the inter-node testing in Fig. 6, and we use a 16-core SMP platform consisting of 4 core i7 quadcore processors (Nehalem 920). Fig. 7 shows that our inter-core parallelization exhibits excellent speedup up to 16 worker

threads with over 13 speedup for both datasets. Namely, the speedup is material characteristic independent.

Finally, we test the effectiveness of our locality optimization approach. Fig. 8 compares the performance with and without Z curve based locality optimization for dataset 1 of different sizes: 1, 2, 5, 8 million atoms, respectively, on a single core i7 processor with 12GB memory. The figure demonstrates the effectiveness of space-filling Z curve in increasing data locality, which improves the performance up to 21%.

6. Conclusion

This paper studied molecular fragment analysis using hierarchical parallelization to harvest computing power of multi-core cluster. We have combined three approaches: (1) inter-node parallelization via spatial decomposition and hook-and-contract algorithm; (2) inter-core parallelization via master-and-worker algorithm; (3) locality optimization based on space-filling curve to improve memory accessing. Experiments showed that our proposed scheme achieves almost linear inter-node strong scalability up to 50 million-atom sized molecular graph up to 32 computing nodes, and over 13 inter-core speedup on 16 cores. Also experiments demonstrate the effectiveness of locality optimization on performance enhancement. However, the inter-node performance degrades when the number of computing nodes exceeds 64, which suggest the need for a better inter-node merging algorithm.

7. Acknowledgement

This work was supported by NSF-PetaApps/EMT, DOE-SciDAC/SciDAC-e/BES/EFRC, and DTRA.

References

- [1] A. Strachan, A. C. T. van Duin, D. Chakraborty, S. Dasgupta, and W. A. Goddard, "Shock waves in high-energy materials: The initial chemical events in nitramine rdx," *Phys. Rev. Lett.*, vol. 91, p. 098301, 2003.
- [2] A. Nakano, R. K. Kalia, K. ichi Nomura, A. Sharma, P. Vashishta, F. Shimajo, A. C. T. van Duin, W. A. Goddard, R. Biswas, D. Srivastava, and L. H. Yang, "De novo ultrascale atomistic simulations on high-end parallel supercomputers," *Int'l J. High Performance Comput. Appl.*, vol. 22, pp. 113–128, 2008.
- [3] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings," *Int'l J. Parallel Program.*, vol. 29, pp. 217–247, 2001.
- [4] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, "Computing connected components on parallel computers," *Commun. ACM*, vol. 22, pp. 461–464, 1979.
- [5] F. Y. Chin, J. Lam, and I.-N. Chen, "Efficient parallel algorithms for some graph problems," *Commun. ACM*, vol. 25, pp. 659–665, 1982.
- [6] D. B. Johnson and P. Metaxas, "Connected components in $o(\log^{3/2} v)$ parallel time for the crew pram (extended abstract)," in *Proceedings of the 32nd annual symposium on Foundations of computer science*, ser. SFCS '91. Washington, DC, USA: IEEE Computer Society, 1991, pp. 688–697.
- [7] K. W. Chong and T. W. Lam, "Finding connected components in $o(\log n \log \log n)$ time on the crew pram," *Journal of Algorithms*, vol. 18, pp. 378–402, 1995.
- [8] C. Kruskal, L. Rudolph, and M. Snir, "A complexity theory of efficient parallel algorithms," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science. Springer Berlin, 1988, vol. 317, pp. 333–346.
- [9] A. Nakano, "Multiresolution load balancing in curved space: the wavelet representation," *Concurrency: Practice and Experience*, vol. 11, pp. 343–353, 1999.
- [10] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," *IEEE Transactions on Knowledge and Data Engineering*, vol. 13, pp. 124–141, 2001.

Accelerating the Hough Transform with CUDA on Graphics Processing Units

Su Chen and Hai Jiang

Department of Computer Science, Arkansas State University, Jonesboro, AR 72467, USA

Abstract—Circle detection has been widely applied in image processing applications. Hough transform, the most popular method of shape detection, normally takes a long time to achieve reasonable results, especially for large images. Such performance makes it almost impossible to conduct real-time image processing with sequential algorithms on community computers. Recently, NVIDIA developed CUDA programming paradigm to explore the tremendous computational power for operations on vectors, matrices and high-dimensional matrices. In this paper, two Hough transform algorithms are designed to run on both CPU and GPU computing platforms. Experimental results indicate that the better Hough transform on GPUs can achieve up to 400 times speedup over the version on CPU. With other efficient image scaling algorithms, real-time circle extraction can be achieved with GPU support.

Keywords: Hough Transform, Graphics Processing Unit, CUDA

1. Introduction

Hough transform is a popular technique for feature extraction in image processing and computer vision. This concept was first proposed to detect straight lines [4] and was later generalized into a robust technique to detect the positions and directions for any shapes that are already known [2]. Such scheme was known as generalized Hough transform. Because of its powerful nature on shape recognition, Hough transform also plays an important role in image and object reconstructions. However, the classical Hough transform adopts brute-force approach, which normally takes long execution time to detect shapes with more than two parameters, such as circles and ellipses. Many researchers have been working on optimizations of Hough transform. So far, the execution time of Hough transform to detect shapes with multiple parameters is still intolerable.

Circle detection can be found in many applications in a wide range of academic areas, such as medical image processing [3], [7] and robot vision [10]. Since a circle in $2 - D$ plane has three parameters, the parameter domain should be a $3 - D$ cube, which requires long execution time and large memory capacity. To solve such kind of problem, people try to reduce the dimension of parameter space using specific techniques on certain problems [9]. However a generalized solution has not been accomplished yet.

The new Fermi architecture designed by NVIDIA [5] provides more CUDA cores than previous versions such as G80 and GT200. Also, data cache is provided to speed up each sequential task on GPU. New warp handling strategies on streaming multiprocessors (SM) are designed to manage threads better. Some new features of Fermi architecture can be explored for performance gains.

This paper intends to accelerate Hough-transform-based circle detection with and without parameter space using CUDA technology on GPUs [6]. It makes the following contributions:

- Two sequential Hough transform algorithms are developed for CPU execution.
- GPU versions of Hough transform are deployed for NVIDIA Fermi architecture.
- Detailed experimental results and performance analyses are provided to demonstrate the effectiveness of CUDA acceleration.

The rest of the paper is organized as follows: Section 2 discusses the detail of circle detection using generalized Hough transform. Section 3 introduces the sequential and CUDA parallel algorithms. Section 4 provides performance analyses on both CPU and GPU. Section 5 gives the related work. Finally, our conclusion and future work are described.

2. Hough Transform and Implement on Circle Detection

2.1 Concept and examples of Hough transform

In this paper, we only discuss shapes on $2 - D$ plane. Shapes with two parameters a and b can be represented as a function $f(a, b) = 0$, such as $x + ay + b = 0$ and $x^2 + ax + b - y = 0$ for lines and a special type of parabolas, respectively. While the image domain resides in the $X - Y$ coordinate system, the transformed domain, or parameter domain, should be located in the $A - B$ coordinate system. Fig. 1 and Fig. 2 give rough ideas about Hough transform on lines and circles.

In 1, we can see that in the image domain, there are two straight lines: $y = x + 1$ and $y = -x + 3$. Since we wrote the equations of lines in slope intercept form $y = kx + b$, it is better for us to choose slope k and intercept b for the coordinate axes of the transformed domain. What we are going to do next is to make the points: $(1, 1)$ and $(-1, 3)$

stands out of all points in the parameter domain so that we can “see” them and reconstruct our image using their information. Hough transform on straight line excavates the duality property of dot and line. For example, the equation form of $y = kx + b$ can be rewritten as $b = -xk + y$, which can be seen as the equation form of line in the $K - B$ domain. Given an dot (x_0, y_0) on $y = k_0x + b_0$, we can get $y_0 = k_0x_0 + b_0$, which can be reformed to $b_0 = -x_0k_0 + y_0$, a regular straight line that passes (k_0, b_0) in the $K - B$ domain. Since we have a whole bunch of dots on the line $y = x + 1$, where $k_0 = 1$ and $b_0 = 1$, we will get a cluster consisted by many lines that pass $(1, 1)$ in the $K - B$ domain. For the parameter domain, we set counters for every pixel and increase them by 1 when it was passed by some line. As it is shown in 1, this strategy can make the point $(1, 1)$ and $(-1, 3)$ outstand in the parameter domain. It is noteworthy that when k and b approaches infinity, parameter domain becomes infinite large, which makes the problem impossible to deal with by computers. This problem can be solved by using polar coordinates in the parameter domain, which uses (ρ, θ) as parameters. Usually, we shall specify the range of ρ since it may become very large and take too much memory space.

Switch to detecting circles, since their equation are different from those of lines, the above experience cannot be simply transferred. Actually, if we understand the spirit of Hough transform, we would be able to find several strategies to achieve our goals. The equation of circle can be written as $(x - a)^2 + (y - b)^2 = c^2$, so we set our parameter domain as a 3 - D cube $A - B - C$ following the regulation we stated before. Assuming we have a black point (x_0, y_0) in the image domain, which can be seen as a point on the circumference of circle centered at (x_c, y_c) , we can select all other points as this center point, then r_c can be calculated by: $r_c = \sqrt{(x_c - x_0)^2 + (y_c - y_0)^2}$, hence we get a set of coordinate (x_c, y_c, r_c) corresponding to a point in the parameter domain. We set counters for all points in that 3 - D cube and increases them by 1 when they are “visited” by the calculated (x_c, y_c, r_c) . It can be imagined that for one (x_0, y_0) , there will be a conical surface radiate from the point $(x_0, y_0, 0)$ along the line $f(a, b) : \{a = x_0; b = y_0\}$ in the 3 - D cube. The conical surfaces here are counterparts of the lines in the $K - B$ domain mentioned above. If we deduce back from the parameter domain, we can find that for a real circle, say $(x - a_0)^2 + (y - b_0)^2 = c_0^2$, counter’s value of the corresponding point (a_0, b_0, c_0) in the parameter domain must outstand among its neighbors. We can see this phenomenon clearly in 2, where three circles shaded by intensive noises are found in parameter domain as peaks.

Another good nature of circular Hough transform is we do not have to worry about the infinite slope, but the range of r should still be specified to save space and calculation time.

In real applications, we should not always follow the traditional algorithm but need to find better ways to implement

it. There are several mapping strategies between the image domain and the parameter domain to make right points stand out. Two strategies of circular Hough transform and their relative merits will be discussed.

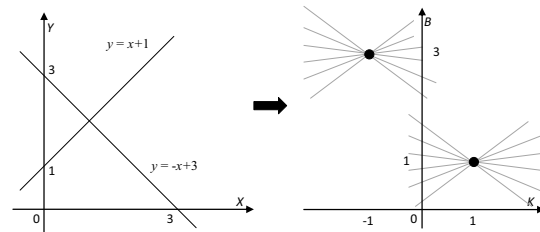


Fig. 1: Hough transform on lines

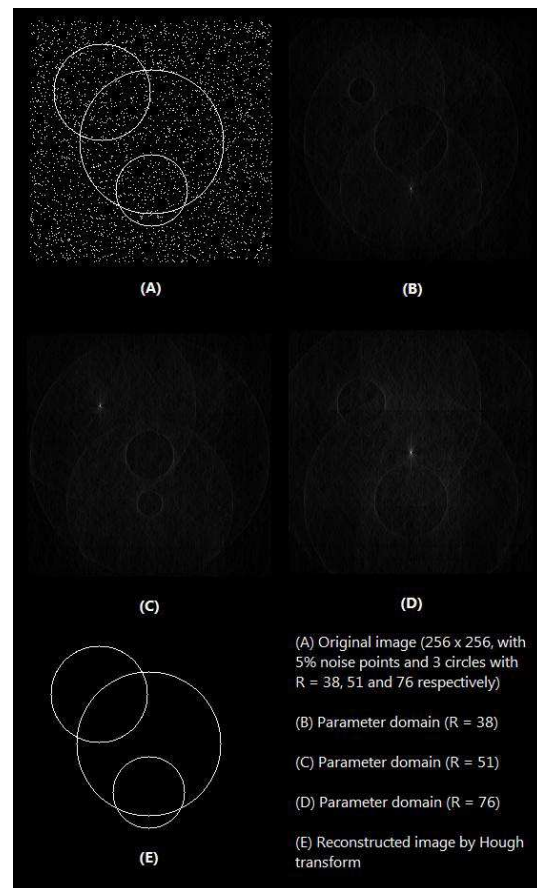


Fig. 2: Hough transform on circles

2.2 Straightforward Mapping Strategy

The straightforward mapping strategy basically follows the steps of the circular Hough transform that is described in Section 2.1. It has some advantages in circle detection. First, it can locate positions in parameter domain very quickly. Since x and y are loop variables residing in registers, r is

the only thing needed to be calculated by the formula $r = \sqrt{(x_0 - x)^2 + (y_0 - y)^2}$. Therefore, the time complexity in the loop kernel will be reduced. Second, in practical applications, images are sometimes pre-processed into binary format for Hough transform. In this case, the straightforward strategy can simply ignore background pixels and only work on the meaningful ones. Those background pixels might be black or white and their value should be 0. Then, it is unnecessary to add them up for any position in the parameter domain. Eventually, this will reduce computing time significantly when background dominates total image area. On the other hand, space taken by the parameter domain grows in $O(n^3)$ in circle detection. This becomes intolerable when n becomes large. By confining the range of r and conducting more rounds of calculation, the parameter domain buffer can be reduced linearly as partitioning frequency increases. With such a strategy, this problem can be solved to some extent. However, in CPU, partitioning r too many times will harm the cache coherence severely and slow down the execution. Concrete flowchart for sequential algorithm for this mapping strategy is given in the left portion of Fig. 3.

2.3 Inverse-checking mapping strategy

Inverse-checking mapping strategy completely solves the space saving problem. It checks back from the parameter domain to the image domain and see if there exists a circle or not. If the number of accumulated pixels for a potential center pixel indicates that there exists a circle, the parameter will be saved to a certain array temporarily or printed out directly. Otherwise, the calculated one will be discarded instantly.

Although this mapping strategy is space efficient, it contains some limitations and shortcomings. First, this method needs more calculation time to find exact positions for pixels on virtual circles. To locate the circumference for given x , y and r , we have to use *sine* and *cosine* functions, which are more time-consuming. Vectors might help save some time on this operation. Second, this mapping algorithm cannot circumvent the background points and have to calculate them all. So the execution time of this algorithm is predictable. This might be a good feature for parallel computing but a disaster for binary images since it cannot take advantage of the implied information in images. Flowchart for sequential algorithm for this mapping strategy is given in the right portion of Fig. 3.

3. CUDA Version Code Design

NVIDIA GPUs and CUDA provide new computing platform for Hough transform. From the hardware perspective, the Tesla C2050 or C2070 has 16 streaming multiprocessors (SM) and each SM owns 32 CUDA cores. From the software perspective, CUDA allows user to generate thread blocks whose number is no larger than 65536 (block arrangement can be 1D, 2D or 3D) and the thread number in block should

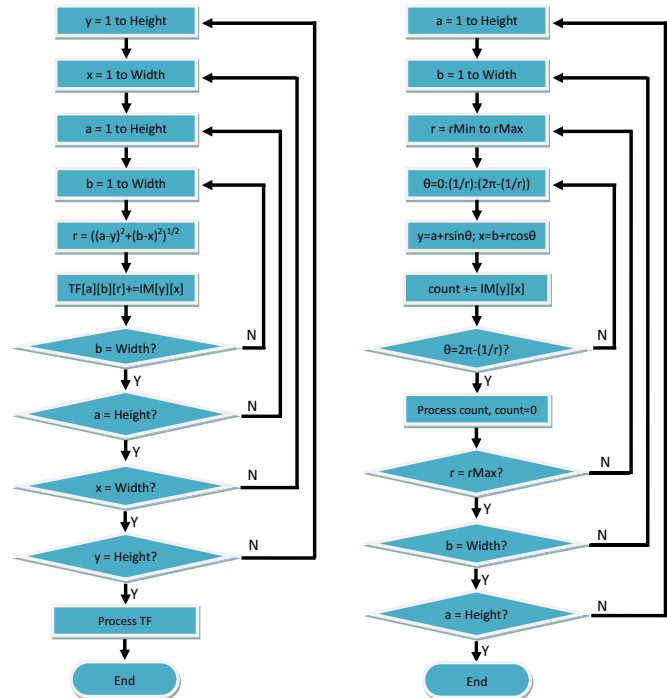


Fig. 3: Flow charts for straight-forward and inverse-checking mapping strategies

not exceed 1024. This friendly design eases programming since each pixel in the image domain can be mapped onto one hardware thread in GPU. Most our test images are only 512×512 and much smaller than the hardware capacity 65536×1024 . Also, since calculations in the Hough transform are totally independent, CUDA and GPU become the ideal platforms.

The latest version of CUDA provides asynchronous data copy functions so that the overlapping of computation and communication becomes possible. This design can hide file transfer latency between host and device memory and keep GPU busy with the actual calculations in Hough transform.

3.1 CUDA Design for Straightforward Strategy

For straightforward mapping strategy, since workloads for different pixels in image domain are independent with each other, they can be mapped onto different threads. From the programming perspective, it can be simply realized by unrolling the outmost two loops of the sequential code and adjusting some necessary dependencies. The workflow using CUDA is shown on the left side of Fig. 4.

Four major steps are involved. First, the original image is transferred from host memory to device memory. Then the GPU starts the transformation process. After the transformation is completed, GPU threads will search for intensive points in transformed domain and output them as circles in a new image. In the end, the new image will be transferred

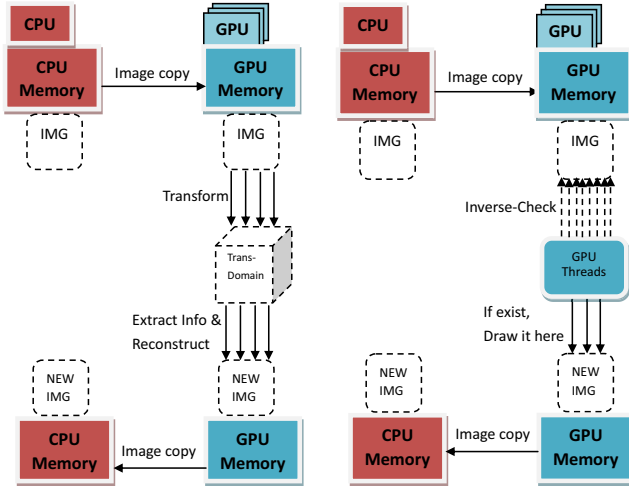


Fig. 4: Straight-forward & Inverse-checking strategy on GPU

back to CPU and print out to validate the whole process.

3.2 CUDA Design for Inverse-checking Strategy

For the inverse-checking mapping strategy, it is impossible to map each set of x , y and r to separate threads since the number is too big. Here we still map each pair of x and y on different threads, the work load for which are expected to be the same. A noteworthy thing is that one should be cautious if he wants to continue partitioning the dimension of r , since if the r -dimension is partitioned into some intervals like $[1, \frac{r}{2}]$ and $[\frac{r}{2} + 1, r]$, the expected work load will differ. This is because larger r will search a bigger circle and thus incur heavier workload. The workflow of inverse-checking strategy is given on the right of in Fig. 4.

Obvious difference on the GPU side can be observed between two graphs in Fig. 4. The data copy procedures between memories are the same, but the ways that the GPU gets the new image are different. The inverse-checking strategy shown in right side of Fig. 4 does not bother with a 3-dimension buffer. In addition, a fewer threads may be able to detect circles and output them to the new image. Usually, the number of circles is not proportional to that of threads. However, the imbalanced workload brought by this can be ignored.

4. Experimental Results and Performance Analyses

Both straightforward and inverse-checking mapping algorithms have been tested on a machine with two Intel Xeon E5504 Quad-Core CPUs (2.00GHz, 4MB cache) and two NVIDIA Tesla 20-Series C2050 GPUs.

4.1 Experimental Results

The relationship between the side length of an image and its execution time on CPU and GPU with above two aforementioned strategies are given in Fig. 5, where CPU1 and GPU1 stand for the straightforward mapping algorithm, whereas CPU2 and GPU2 represent the inverse-checking mapping algorithm. The corresponding speedup graph is also shown in Fig. 6.

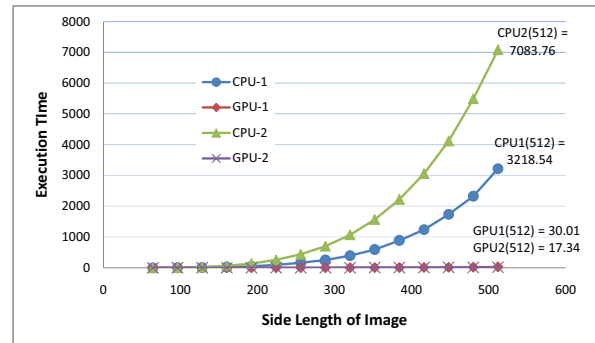


Fig. 5: Execution time comparison between CPU and GPU versions

Compared to straightforward mapping algorithm, the inverse-checking mapping algorithm can benefit GPU but hurt CPU version's performance. Several factors are responsible for this.

First, on the CPU side, the inverse-checking algorithm incurs more computations, which fundamentally increase the execution time. Even though the parameter domain requires much memory space, most time CPU can handle this situation well unless the image size is too large for memory and cache. However, on the GPU side, large space for parameter domain might cause performance degradation quickly since threads are grouped into blocks and it is hard for them to communicate with each other. Cache within a block is local and cannot be seen by threads from other blocks. This design of Fermi architecture does not help Hough transform problem. However, massive threads can still help GPU achieve performance gains by hiding the memory access latency.

Second, for the inverse-checking algorithm, the sizes of input and output images are not very large for current cache in Fermi architecture. Although the total cache is partitioned into small independent pieces when blocks are generated, a good cache hit rate still can be achieved. The reason is that the size of the input image is $O(n^2)$ and this is the only space for accesses. This property improves the GPU's performance dramatically. However, the selection of grid and block dimension also has a great influence on overall performance.

GPU acceleration results have been shown in Fig. 6 illustrates how many times can be achieved by the latest GPU

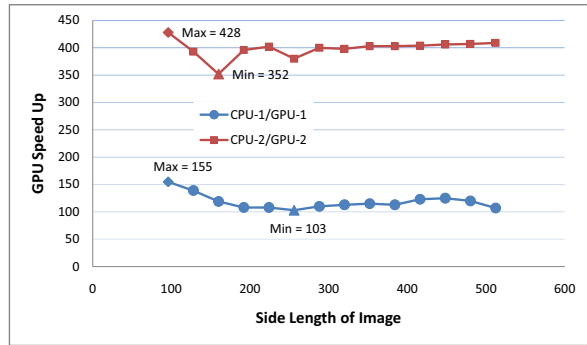


Fig. 6: GPU Speedups for both mapping algorithms

architecture. The straightforward algorithm of maintaining the parameter domain buffer can achieve about 100 times speedup, whereas the inverse-checking approach can reach about 400 times speedup. The latter one does not have to keep a large parameter domain. The speedup curves in Fig. 6 vary as the image size increases.

4.2 Performance Analyses

Although experimental results in Figs. 5 and 6 have demonstrated that GPU versions can beat CPU ones for up to 400 times. However, the internal reasons about why GPU can accelerate Hough transform so well and why inverse-checking algorithm can exhibit better performance need further investigation. NVIDIA Fermi architecture plays the critical role.

It is clear that the maximum speedup is achieved when image size is relatively small such as 96×96 . In fact, images with 64×64 size have been tested as well. However, GPU version execution was too fast to catch the accurate result for comparison. Such significant speedup in small problem size cases are caused by data cache capacity in GPUs. It can be observed that after size length of 96, curves go down quickly. The main reason is that cache hit rate drops quickly as the image size increases.

Another factor that might influence cache hit rate largely is the selection of block dimension. Since each pixel is mapped onto one separated thread, the choosing of block dimension will simultaneously decide the total block number. When more blocks are generated, the cache size assigned to each block will decrease. Therefore, the cache hit rate within blocks will be hurt. However, putting too many threads (1024 threads can be the limit within one block) into one block is not a wise choice as well. All these threads will be handled by one SM in Fermi architecture. When the limit is reached, extra threads will put into queues. Traditionally, one SM will handle at least one block and the maximum number of threads it could handle concurrently is fixed. When block size is big, the optimal thread number for one SM will be harder to achieve. Smaller block size such as 32, 64 and 128,

are usually good selections. Figs. 7 and 8 illustrate how the best execution times can be achieved for different settings of block dimensions as the problem size increases.

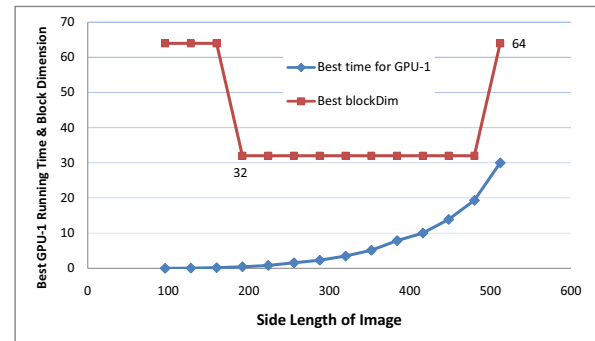


Fig. 7: GPU version's best execution time and block dimension selection with straightforward algorithm

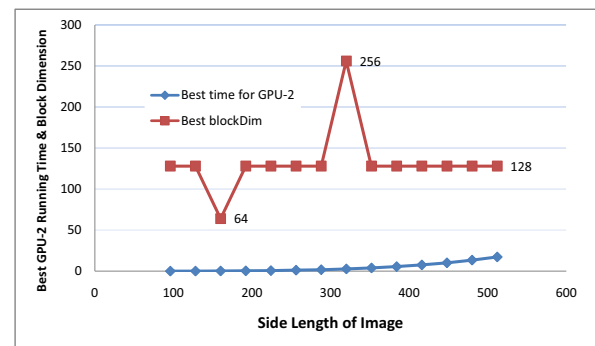


Fig. 8: GPU version's best execution time and block dimension selection with inverse-checking algorithm

Figs. 9 and 10 demonstrate comprehensive results for both algorithms on different block and input sizes. It is clear that the straightforward algorithm's performance is not really stable as the image size increases, whereas the ones for the inverse-checking algorithm maintain very good shape and almost increase exactly by $O(n^4)$. In Fig. 9, $54.08 \div 2.73 = 19.81$ and in Fig. 10, $60.19 \div 3.77 = 15.96$. Time increase from 256 to 512 should be $(\frac{512}{256})^4 = 16$, which matches Fig. 10 very well. For Fig. 9, however, the value 19.81 exceeds theoretical value 16 a lot. This indicates the massive memory access in GPU will slow down execution significantly. Another noteworthy issue is that, for the inverse-checking algorithm, which is computationally intensive in GPU, curves for cases with block dimensions equal to and larger than 64 are similar to each other, i.e., they all exhibit good performances. Therefore, in computationally intensive cases, these dimensions for block might be better choices. Although NVIDIA suggests 64 or its multiple

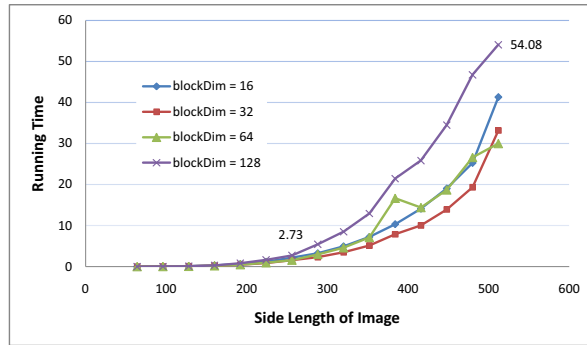


Fig. 9: Execution time of straightforward algorithm with different block dimensions

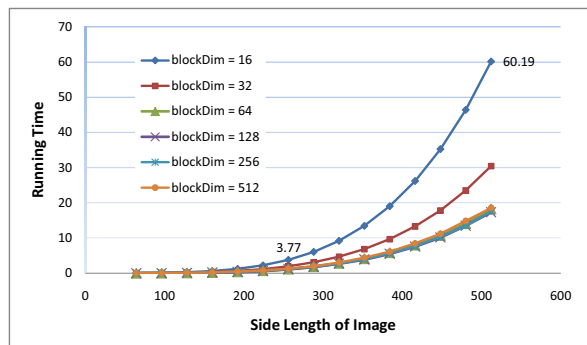


Fig. 10: Execution time of inverse-checking algorithm with different block dimensions

number threads for one block, in straightforward algorithm, a selection of 32 threads for a block is also the best choice sometimes.

4.3 Possibility for Real-time Processing

It is clear that the inverse-checking algorithm performs better on GPUs. This might support real-time circle detection. Although nearly every application requires faster processing speed, the video processing is especially demanding on this. Assume the frame rate of a video is 25 frames per second. To detect circles in real time, the total transferring time (t_{trans}) and processing (t_{proc}) time for one frame should not exceed 0.04 second. As we mentioned before, t_{trans} can be ignored when asynchronous data transfer technology is applied. On the other hand, t_{proc} might include video decompression time (t_{decomp}), image scaling time (t_{scale}) and circle detection time (t_{hough}). Since real-time compression and image scaling can be achieved, t_{decomp} and t_{scale} just exhibit as processing delays when computation/communication overlapping scheme is used. Can we reduce Hough transform time t_{hough} to 0.04 second or less? Based on our experimental results, when the image size is around 96×96 , t_{hough} is between 0.02 and 0.03. Thus,

if this resolution is tolerable for circle detection, real-time processing can be achieved. In many video processing applications, just regularly sampled frames will be processed. The proposed scheme can support larger images.

5. Related Work

Hough transform was first introduced to detect lines by Paul Hough in 1959 [4]. After that, Richard Duda and Peter Hart (1972) invented generalized Hough transform [2], which became popular in computer vision field after Dana Ballard published his paper in 1981 [1]. Since then, this technique has played an important role in shape-based feature extraction.

Jaroslav Borovicka (2003) wrote a paper about circle detection using Hough transform [8]. As a popular tool in image processing, Hough transform is included in toolboxes of Matlab by MathWorks, but the functionality is limited to line detection. In 2010, CUVILib from NVIDIA developed a library for Hough transform on straight lines.

6. Conclusions and Future Work

This paper takes the advantage of NVIDIA GPU's computability and developed proper methods to parallelize Hough transform on circle detection. Compared to sequential C code, CUDA code can achieve up to 400 speedups with inverse-checking approaches on Fermi architecture. With such a high acceleration rate, real-time circle detection using non-modified Hough transform becomes possible. As a computationally intensive problem with relatively small I/O, Hough transform exploited GPU's computational power thoroughly. In future, we plan to detect more types of shapes using Hough transform and latest features of GPU architecture and CUDA programming paradigm will be explored and put into use.

References

- [1] D.H. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, 13, No.2:111–222, 1981.
- [2] R. O. Duda and P. E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15:11–15, 1972.
- [3] S. Eom, R. Bise, and T. Kanade. Detection of hematopoietic stem cells in microscopy images using a bank of ring filters. In *The IEEE International Symposium on Biomedical Imaging*, 2010.
- [4] Paul. V. C. Hough. Method and means for recognizing complex patterns, 1962.
- [5] Nvidia. Nvidia fermi tuning guide, 2009.
- [6] Nvidia. Nvidia cuda c programming guide 3.2, 2010.
- [7] M. Smereka and I. Duleba. Circular object detection using a modified hough transform. *International Journal of Applied Mathematics and Computer Science*, 18, No. 1:85–91, 2008.
- [8] University of Bristol. U.K. *Circle Detection using Hough Transform*, 2003.
- [9] Y. Xie and Q. Ji. A new efficient ellipse detection method. In *International Conference on Pattern Recognition*, 2002.
- [10] Y Yabuta, H Mizumoto, and S Aarii. Binocular robot vision system with shape recognition. In *International Conference on Control, Automation and Systems*, 2007.

Fast Dot Correlation in Optical Metrology on GPGPUs

Ralf Seidler¹, Andreas Schäfer¹, and Dietmar Fey¹

¹Computer Science 3, Chair of Computer Architecture
Friedrich-Alexander-University Erlangen-Nuremberg, Germany
{ralf.seidler, andreas.schaefer, dietmar.fey}@informatik.uni-erlangen.de

Abstract—*In optical systems image defects of the used lenses are a decisive factor and can significantly impair the accuracy of the measurement results. To measure the defects, the lens is placed in front of a dot mask and a CCD-sensor. When large numbers of lenses need to be inspected, time is a crucial factor. For those measurement systems employing standard PCs a viable option for image processing is to make use of the parallel computing power of GPUs by offloading compute intensive tasks to them.*

In this paper we present fast and efficient parallel algorithms for finding the dots on the sensor with sub-pixel accuracy and allocating them to their expected dot positions. For that purpose, we propose a cellular automaton based approach which we have evaluated on NVIDIA Fermi GPUs. For a resolution of 16 megapixels, resulting in about 30000 dots and 120 iterations, we have achieved 25 frames per second. Compared to previous work, we show that even difficult benchmark instances can be solved with a speedup of about 90.

Keywords: parallel algorithms; optical measurement; cellular automata; GPGPU; CUDA

1. Introduction

The quality of optical systems such as cameras, spy-glasses and telescopes depends on the accuracy and quality of the used components. The creation of such devices has made large steps towards mass-production and miniaturization. Image defects impair the usefulness of these devices. When knowing the exact degree of the image defects, the optical distortion can be corrected resulting in images of higher precision. For that the industry needs fast and reliable quality control in their production processes.

Examples of non-destructive measurement of materials are the optical 3D-metrology based on stereo-camera systems [1], whitelight interferometry [2] or the usage of x-ray or supersonic diagnostics. In addition to these methods another destruction-free approach can be implemented by using an optical dissection array.

In the wide area of measurement and testing technology huge amounts of data need to be entered into the system while being recorded at the same time. For these applications the use of multi-core processors seems feasible. The described metrologies are often implemented in production processes as means of quality control. Hence the analysis of

the data has to meet real-time constraints making a parallel computation of data necessary. In most cases the problem space can be decomposed geometrically in smaller parts to be worked in parallel. Accordingly, fast dot correlation in optical metrology on GPGPUs is within the application field of data-parallel computation.

For data-parallel applications, hardware-accelerators are suitable. The most promising being GPUs. They can be programmed flexibly and are currently used successfully for a variety of jobs in high-performance computing [3], offering a high FLOPs per dollar ratio. Previous works show that graphics processing units can be efficiently used to accelerate many computational tasks such as database applications using SQLite [4], physical simulation based on monte-carlo methods [5], cellular automata simulation and visualization [6], high performance stencil code computation [7] as well as improving the speed of the image processing toolbox for the well known Matlab software [8].

We will examine how GPUs perform when they are used for a real-time application for optical measurement techniques. For that we introduce a cellular automaton [9] based dot correlation algorithm designed for massively parallel architectures. All necessary computations, including the dot exploration, are done on the graphics hardware which serves as a fast co-processor for the CPU. This approach minimizes the need for communication between CPU and GPU.

2. Background and Related Work

In optical metrologies one of the most common tasks is to measure and evaluate the quality of an optical device, e.g. a lens, a mirror or a prism. The lenses and mirrors of a telescope for astronomic application must be as perfect as possible. To determine the quality and evenness of such a large device, extensive measurements are necessary. If this measurement process would take several seconds, the complete scan of the device could take several hours or even days. Contact lenses which are produced in very high numbers also have to meet very high quality standards. During the necessary quality control of lenses the speed of the measurement is a crucial factor for increasing the production throughput or to start the rework process (e.g. polishing) earlier.

An optical phase distortion sensor initially proposed by Hartmann (hereafter abbreviated as H-sensor, an advanced version of such a sensor can be found in [10]) can be used

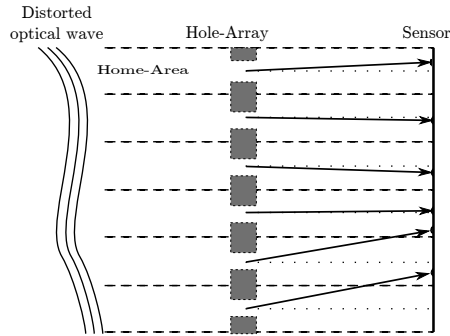


Fig. 1

EVALUATION OF A DISTORTED LENS USING AN OPTICAL SENSOR WITH
A DISSECTION ARRAY ELEMENT

to measure a distorted lens. To generate the required dots, an optical dissecting array element (e.g. with holes) is installed between the lens and an image-sensor. The resulting image shows a dot-array, where every dot has a predefined location on the sensor given by the focus of one hole when a plane wave is measured. This is illustrated by the dotted vertical lines in Figure 1. It also shows that one part of the optical wave that is to be measured, belonging to one particular hole, is called home-area. The dots can be outside of their respective home-area. In this case they need to be correlated with their expected position.

In [10] a measurement technique for an H-Sensor is introduced and the dot-exploration process is discussed. A sensor of 640×480 pixels is being used. The micro-lens array has a dimension of 15×15 lenses. With a Pentium IV 800 MHz, they obtained a measurement speed of approximately 22 ms. This is achieved by partitioning the image into 15×15 parts, corresponding to the home-areas. Over these parts a middle-point computation algorithm is performed. The middle-point σ of an area $A \in \mathbb{N}^{M \times N}$ of size $M \times N$ in a gray-scale image can be described as

$$\sigma_x = \frac{\sum_{i,j \in A} i \cdot \xi_{i,j}}{\sum_{i,j \in A} \xi_{i,j}}, \quad \sigma_y = \frac{\sum_{i,j \in A} j \cdot \xi_{i,j}}{\sum_{i,j \in A} \xi_{i,j}} \quad (1)$$

with $\xi_{i,j}$ being the gray-scale value of the pixel at position (i, j) of the image.

A fast computation can be achieved, because there is no data-dependency between the areas. This algorithm is the state of the art for these problems and will be the basis for our experimental comparison. Modern H-Sensors use hole-arrays with up to 250×150 holes, based on an image-sensor of 16 megapixels. With that the processing of one image would take approximately one second on today's standard PCs.

When using H-sensors the dots induced onto the image-sensor may leave the predefined search-location. In that case, the proposed search-algorithm in [10] would not find all dots, rendering the measurement useless. To avoid that a dot correlation is needed. After having correlated the found dots with their corresponding rays, a complex post-processing

step follows. This post-processing step is mostly harder to compute than the correlation, but can be implemented efficiently on graphics cards. The focus of our work is to do the data-reduction achieved by the dot exploration and correlation. After that, the data can be held on the graphics card for further post-processing computations.

The described method can operate on gray-scale images. Each pixel of such image can be 8, 16 or 32 bits in depth, where a maximum resolution of 16 megapixels is expected. This results in up to 64 megabytes of memory per image. We aim for a frame rate of at least 25 fps for the complete correlation, since this is mostly the industrial demand due to the actual optical sensor ability of delivering at least 25 fps.

3. CUDA and the GF100-Architecture

Before introducing the architecture, it is advisable to examine the CUDA programming model. CUDA stands for Compute Unified Device Architecture and was introduced in 2007 by NVIDIA [11]. It offers the ability to offload compute intensive tasks from the CPU, called `host`, to the graphics hardware, called `device` in CUDA context.

CUDA threads are arranged in so called thread-blocks with up to 1024 threads each. Each thread has access to his own registers, a memory shared among the threads of a thread-block and global memory. The data residing in global memory is accessible by all thread-blocks and the host machine. This is used to transfer data between the host and the device memory. Another important feature – the so called `atomic` functions – allow an exclusive access of one thread to a memory address in global memory.

A graphics processing unit consists of a number of multi-processors (MPs). Each of them has a number of processing elements (scalar processors), some internal memory and special function units (SFUs). To execute a program, one thread-block is mapped to a multiprocessor and executed in SIMD-like manner. Always taking 32 threads (called a warp) onto the execution units, each thread executes the same operation. Nevertheless, branching is possible where each non-active thread must be masked out. So in the worst case, the potentially parallel code is serialized.

NVIDIA's GF100 architecture codename "Fermi" offers several benefits, such as cached global memory and a large on-chip memory, used as L1-Cache or shared-memory per multiprocessor [12]. As described in [13] the architecture can be utilized effectively, if the program exhibits a high L1-Cache hit-rate. Each multiprocessor has 32 processing elements compared to 8 in older generations. One multiprocessor can now execute more than one thread-block, resulting in a more concurrent behavior of accesses to the register-file, shared memory and caches. As described in [14] older programs can still be run on the new devices.

The cache for global memory and the shared memory are physically the same. There are 64 kB of memory available per MP. This can be dynamically adjusted per kernel to

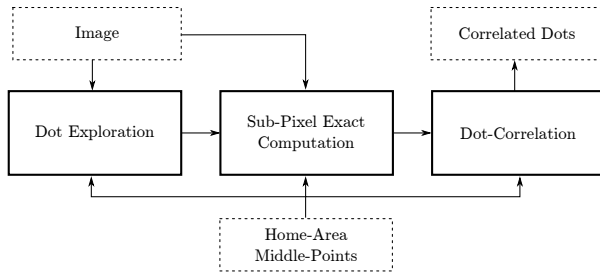


Fig. 2

THREE-STEPS MODEL OF THE PROPOSED ALGORITHMS

either 16 kB L1-Cache/48 kB Shared Memory or 48 kB L1/16 kB shared. It is advisable to adjust the size to the most beneficial setting for that particular use-case. A GPU acts like a CRCW-PRAM [15], [16] (Concurrent Read Concurrent Write Parallel Random Access Machine). So every thread of a program can access any global memory address for read or write at any time. If more than one thread want to write to the same address, it is undetermined which thread wins, but one is guaranteed to. The communication of several threads of one thread-block is needed to achieve a high throughput. Nevertheless, no global barrier technique is available. Especially in iterative algorithms, such synchronization has to be done by repeatedly starting the same kernel, inflicting several microseconds of waiting time [17].

For our tests we used a system with an Intel Core i7 920 with four cores, running at 2.66 GHz. The graphics card is an NVIDIA Geforce GTX 480, based on the above mentioned GF100 chipset.

4. Proposed Algorithms

The operations described here can be illustrated as a three-step-model and are shown in Figure 2. We call one partition of the image that surrounds an expected position of a ray *home-area*. The first step is to explore where dots are present on the image. This is followed by the sub-pixel exact computation of the found dots. After that the found dots are correlated to their corresponding home-area. In all three steps, the home-area middle points (the expected positions of the dots) are needed.

4.1 Dot Exploration

To travel through the image one line at a time and determine if the currently analyzed pixel is a maximum point would be a simple serial dot exploration algorithm. This serial approach can be adapted for a parallel implementation.

Figure 3 displays the data-structure that every home-area has. It illustrates that the middle-point and a place where the exact dot can be stored are needed. Depending on the distortion of the rays, more than one old dot can be present in one home-area. These dots need to be found and stored for further correlation. To keep track of them an index variable is created.

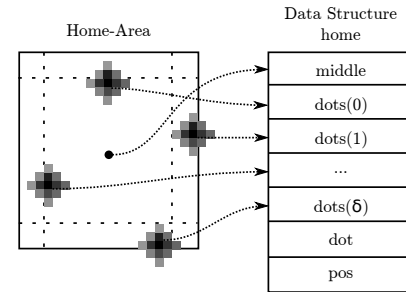


Fig. 3

DOT EXPLORATION PER HOME-AREA. THE DATA-STRUCTURE *homes* CONSISTS OF THE MIDDLE POINT OF THE HOME-AREA, THE FOUND *dots*, A CORRELATED *dot* AND THE NUMBER OF FOUND DOTS *pos*. THE COMPONENTS ARE ACCESSIBLE VIA *homes.{middle,dots(i),dot,pos}*.

Algorithm 1 shows the dot exploration phase. First the image is partitioned into overlapping boxes – the home-areas mentioned above – that are computed in parallel. Their centroids would be the expected dot positions in an ideal case. In each of these partitions, local intensity maxima need to be found, indicating that there is one dot of a ray found. Due to the noise present in the image, these local minima can be present in almost every position. Because of that, a predefined threshold value *th* is introduced. When the value of the particular pixel is higher than *th*, its neighbors need to be taken into account, to determine whether it is the maximum or not. The number of found spots is called δ .

If a pixel is an area maximum, its coordinates need to be inserted into the data structure *home*, so that it can be accessed for future computations. Every pixel can be computed independently from the others and thus in parallel. Therefore we need to have an EREW-PRAM (Exclusive Read Exclusive Write-PRAM) [15] shared memory. We require a mechanism of incrementing a shared variable by one processing unit, determining the address of the found dot to be stored in the shared memory *home*. To do that we have *home.pos* included that points towards the next free cell of the data structure.

4.2 Sub-Pixel Exact Computation

After having found the correct pixel, it is time to compute the sub-pixel exact position of the point. Every home-area can be computed in parallel. The list of found dots of each home-area is computed in serial. To do that a small pixel-area around the found dot is taken into account ($rad(dot)$). A common value of that radius is five pixels around a center point. With $A = \{(i,j) | (i,j) \in rad(dot)\}$ in (1) the exact middle point of that dot can be computed.

4.3 Dot Correlation

Having found all dots within each home-area, we now need to correlate them with the home-area they belong to. The initial task is to find dots that are not shifted beyond the borders of their assigned home-areas. Based on these

Algorithm 1 Dot Exploration

```

1: procedure EXPLORE(img, dimx, dimy, th)
2:   Share home
3:   for all  $i \in \text{dim}_x, j \in \text{dim}_y$  pardo
4:      $o \leftarrow \text{img}(i, j)$ 
5:     if  $o > th$  then
6:        $n \leftarrow \text{img}(i, j - 1)$ 
7:        $e \leftarrow \text{img}(i + 1, j)$ 
8:        $s \leftarrow \text{img}(i, j + 1)$ 
9:        $w \leftarrow \text{img}(i - 1, j)$ 
10:      if  $o > (n, e, s, w)$  then
11:         $p \leftarrow \text{inc}(\text{home.pos})$ 
12:         $\text{home.dot}(p) \leftarrow (i, j)$ 
13:      end if
14:    end if
15:  end for
16:  return home
17: end procedure

```

reference points, we begin to iteratively assign dots. We introduce a set called *correct*, to insert all exact correlated dots into. Since dots may be shifted to such an extent that they reside in a neighboring home-area, or several home-areas away, and given the rather smooth nature of the shifting distortion, it is impossible to decide whether a dot is correctly correlated from just a local view of the dots. Rather, a global view is required to determine the area of reference. We assume that some dots in the middle of the image are not shifted out of their home-areas and insert them into *correct*.

After this initial assignment, the remaining home-areas not yet belonging to *correct* and their corresponding dots have to be assigned to the set *incorrect*. Algorithm 2 describes the dot correlation algorithm. Every home-area becomes one cell of a cellular automaton. Beginning with the correct dots, all their neighboring dots become active and evaluate their dot positions by taking the correct home-areas in cardinal direction into account. For instance, we assume that the home-area (i,j) is active, because their right neighbor is correct. The two before last home-area dots are considered in such a way that a vector is created, leading to the approximated position p_{app} of the next dot.

$$p_{app}(i, j) = p(i + 1, j) + (p(i + 1, j) - p(i + 2, j)) \quad (2)$$

Figure 4 illustrates this situation. After having found the approximated area of the dot, the next step is to find its exact position. To do that a search area around p_{app} is introduced. Since we know the correct places of the found dots, we only have to search every home-area adjacent to the found area. The worst case scenario being that – because the radius is always smaller than the size of the home-area – four home-areas need to be scanned. If there is more than one dot found within the search radius, the nearest dot to the approximated position is chosen.

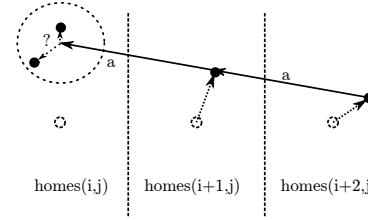


Fig. 4

ESTIMATING THE NEW DOT

Algorithm 2 Dot correlation

```

1: procedure CORRELATE(homes, correct, incorrect)
2:   while not all dots are correct do
3:     for all  $\text{homes}(i, j) \in \text{incorrect}$  pardo
4:       if  $\text{neighbor}(i, j) \in \text{correct}$  then
5:          $n \leftarrow \text{neighbor}(i, j)$ 
6:          $p_n \leftarrow \text{homes}(n).\text{dot}$ 
7:          $p_{nn} \leftarrow \text{homes}(\text{neighbor}(n)).\text{dot}$ 
8:          $p_{app} \leftarrow p_n + (p_n - p_{nn})$ 
9:          $\text{homes}(i, j).\text{dot} \leftarrow \text{SEARCH}(\text{homes}, p_{app})$ 
10:        delete  $(i, j)$  from incorrect
11:         $\text{correct} \leftarrow (i, j)$ 
12:      end if
13:    end for
14:  end while
15: end procedure

```

4.4 Limitations

The described algorithms are a first step towards solving the problems introduced with such an H-Sensor measurement device. They still have some limitations regarding the dots' arrangement to allow a proper correlation. First of all with the existence of noise in the camera image, the found maxima may not be the correct dots. This can only be solved with a proper threshold and a mechanism, to discard dots that are too small. In the correlation phase, finding a suitable starting point is one of the main problems. To achieve this, we introduced a simple middle selection algorithm. If the dots in the middle do not meet these demands the iterative correlation can not start and thus the result would be useless. In our current work, we focus on input images with a relatively small noise present and at least four neighboring dots are selectable by the algorithm. Other limitations are crossing rays or two dots too close to each other. Then the correlation is almost impossible. In that case, a common technique is to mask out some of these dots and compare the resulting image with the one previously taken. That is not an algorithmic, but mechanical approach.

5. Implementation

5.1 Dot Exploration

Implementing the dot exploration on graphics cards can be done in a straightforward fashion. Since we have threads grouped in thread-blocks, this leads to a natural mapping. Every home-area is computed with a thread-block and every

pixel of a home-area is examined by one thread. For that purpose we need to know how large a home-area in the cardinal directions might be. Since new graphics cards have 1024 threads available per thread-block, this leads to a maximum of 32×32 threads, resp. 32×32 pixels. This is enough since industrial requirements are limited to a maximum of 25×25 pixels. Because of the limited resources on the device, the required amount of registers per thread is important as well.

An array with the input-image, its dimensions and the threshold value serve as input parameters for the exploration kernel running on the GPU. Accordingly, an array with the home-area middle points `mid` and the distance between two home-areas `dist` is needed. An array where the detected dots can be stored is allocated as output. Each thread has to determine which of the pixel-coordinates in the image it has to process. For that the pixel id is determined with (3).

$$pixelid = threadIdx + (mid - dist) \quad (3)$$

Every thread loads the value of its corresponding pixel. If it is larger than the predefined threshold, the direct neighbors are loaded. These loads are not aligned and may result in a non-coalesced memory access. In older GPU generations, these could be improved with textures. In the recent generation the new caches are faster and larger than the caches of the texture units. In devices with CUDA compute-capability 2.0 the latter are shared among four multiprocessors. Previously they were shared only among two. All sub-partition data can also be stored in the L1-Cache, so when accessing one memory-word for the first time, it is fetched from global memory and transferred to the L1-Cache. Every pixel (excluding the pixels at the borders of the image) is accessed by four threads, while only one has to wait for the data from memory.

Having discussed the load operations necessary, we will now take a closer look at the control flow operations. They do not perform well on GPGPUs, but are necessary for the next step. In that the threads of the thread-block might diverge, because there are only a few threads, that are over the threshold. By letting all of the threads load their neighbors, we tried to avoid branch-diversion. But since that results in a measurably higher computation time, doing the selection process first is a more rewarding approach. After having found a maximum point, we need to implement the EREW-PRAM alike memory transaction. For that, we need a shared data structure for the home-area and one for the array-index of the found dot. First, this index is set to zero by thread zero and the threads are synchronized. Then the maximum determination described above is initiated. With NVIDIA's chips with compute capability 1.3 and higher it is possible to use `atomicInc` on shared memory. This operation does the same an EREW-PRAM would do. So every thread that has found a maximum does the increment and writes its found dot to the position indicated by the old

index value. To avoid bank-conflicts, we limit the number of dots per home-area to six. This is due to the fact that there is one 32 bits integer as index variable, two 32 bit float values as coordinates of the middle point and for every dot two additional 32 bit float values and one 32 bit value for status flags. This results in only 16 consecutive 32 bit values. So no bank holds more than one value. Now every home-area knows the pixel exact positions of the dots included. Thread zero of all thread-blocks can now write the resulting home-area data-structure to global memory. After all threads are finished, the kernel is stopped and returns.

Due to the data locality, it is not advisable to implement such atomic functionality on global memory. If the transactions were done on global memory, the data would not be guaranteed to be in the caches.

5.2 Sub-Pixel Exact Computation

The next step is to compute the sub-pixel exact positions of the found dots. Since a dot can be found by different home-areas and only one dot is the correct one, the sub-pixel exact computation can be done after the dot correlation phase. Doing the sub-pixel exact computation after the dot correlation phase successfully reduces the runtime. Even though this approach reduces the robustness (since the found dots are only pixel exact when introducing them to the correlation kernel), the reduced runtime – one of our main objectives – is worth it.

In this kernel function, one thread is responsible for one home-area. From the pixel-exact value of the dot, we introduce a bounding box, in which all pixels are considered to be part of the exact dot. This is done because a circle approach does not seem feasible for a graphics card, due to a bad access scheme. When a complete line is considered, the values of the pixels can be cached for further use. One of the major problems here is the non-coalesced memory access of the input image. Every thread accesses a certain area of the image, mostly not in the same line as the other threads in the thread-block. This happens because the dots can differ in their y-coordinate. For a two bytes per pixel image, only 64 pixels can be loaded with one 128 bytes load operation, as it is common in modern graphics cards. Another important factor is that the x-coordinates of the dots are mostly not aligned to 128 bytes accesses. So it is possible that two memory transactions are needed to load every dot. Per multiprocessor, a maximum of 48 kB of L1-Cache is possible, resulting in about 100×100 pixels. If all of these pixels were part of the dots, this would result in only 10×10 threads per thread-block. This would not be enough, since the under-employment of the multiprocessors can not hide the memory-latency properly.

5.3 Dot Correlation

Before we can do the dot correlation, we need to implement some vector arithmetic as well as an efficient search

algorithm. The vector arithmetic is trivial. The search for the correct dot is of more interest. To determine the possible home-areas for the area around the approximated position, its ID is computed with $id = \lfloor p_{app}/dim \rfloor - 1$, where dim corresponds to the exact distance between two home-area middle-points. Since this algorithm is iterative and no efficient synchronization mechanism exists on graphics cards, the kernel needs to be restarted after every iteration. Thus, the first thing needed is the array of the home-area data-structures of the last iteration $home_{old}$, as well as an additional data array $home_{new}$, where the computed result is stored. Also two new character-arrays are necessary. One that describes the current status of each home-area $status_{old}$, the other to hold the new status of this iteration $status_{new}$. These arrays are basically filled with zeros (not correlated) or ones (correlated), describing the status of the corresponding home-area. Other inputs are the dimension of the home-area-arrays and the exact distance between two home-area middle points.

The kernel begins by loading the status of its cell from $status_{old}$ and checks if it has been correlated in a previous iteration. If that is not the case, then the status of the cardinal neighbors is fetched. If two of them are correlated, the computation of the new dot can be done. So it loads the dot from its neighbor $p(i+1, j)$ and its next neighbor $p(i+2, j)$. Now the approximated dot is computed according to Equation (2). With these coordinates the search operation is performed: all home-areas adjacent to the approximated point – meaning they are inside the radius of the search operation – are loaded. All their dots are considered in terms of distance to the approximated point. The one with the smallest distance is chosen and becomes the correct dot of the home-area. Now all home-areas that did an update write a *one* to the status-array $status_{new}$, as well as their resulting data-structure to $home_{new}$ to indicate their correlation. This is repeated iteratively until all dots are correlated.

6. Results

Figure 5 shows the applied dot exploration and correlation algorithms for an example of 12×8 dots. The underlying image has a resolution of 512 pixels with a gray-scale-depth of 16 bits. In Figure 5a the found dots of the dot exploration algorithm are displayed. The correlated result is shown in Figure 5c. The exact home-area middle points, the borders without overlapping and the found dots are shown. Since the correlation algorithm is iterative, the image 5b shows the correlated dots (shaded) after 8 iterations. The final result after 12 iterations is shown in 5c. For larger resolutions, respectively more home-areas, the number of iterations has to be higher.

If there is a dot near a border between two home-areas it is found by all corresponding home-areas, which add it to their list of found dots. This is indicated by a line from the middle to the found dot. During each iteration the correlation of

those home-areas that neighbor already correctly correlated home-areas is done.

For our test purpose we created several test images with different resolutions and different numbers of home-areas per image. In Table 1 the run times for these images on a single core CPU implementation running on an Intel Core i7 920 compared to the run times of an NVIDIA Geforce GTX 480 are displayed. The GPU total time includes the memory-copy time of the image to the device, as well as the time to copy the results back. The correlation time is the time the correlation algorithm takes.

One of the most interesting results to point out in Table 1 is that the correlation kernel takes half of the time of the complete computation, but only a small part of the cells is updated per iteration. This leads to the idea, to find an efficient implementation of an update-scheme which we call "parallel wave propagation update". If something like that were available, it would be highly beneficial for this algorithm and others which we found during our research, e.g. parallel path planning algorithms described in [18].

We estimated the highest achievable number of lattice updates per second (LUPS), computed as $\frac{M \cdot N \cdot iter}{time}$. For a resolution of 4896×3280 pixels and 242×162 dots, we observe 245.4 MLUPS – $(242 \cdot 162 \cdot 121)/0.01933 \text{ ms}$ – for the correlation phase. This very high value is possible since the graphics card is under-utilized with the small number of home-areas in the other examples. Here the GPU can use its full potential: even for the complete computation, with memory copies this is still 120.6 MLUPS. Compared to the run times of the CPU, this is a speed-up of almost 20.

The described algorithms are highly efficient and utilize the graphics card very efficiently for high resolutions and a large amount of dots. The demanded real-time capability is also achievable. With the tool `computeProf` from NVIDIA it is possible to profile a CUDA application. With this tool it was possible to determine, that the L1-Cache hit-rate for the dot-exploration algorithm is about 60%, that being the reason for the high data throughput per home-area. For the dot-correlation algorithm we achieved an L1-Cache hit-rate of over 90%.

We approximate the dot exploration time (without correlation) of [10] to take 3.5 seconds for a 16 megapixels image. With our implementation, we can achieve a 175 times faster dot exploration. With our dot-correlation this is still 90 times faster.

7. Conclusion and Outlook

In this paper we described a fast dot exploration and correlation algorithm for speeding up optical measurement methods. We implemented an efficient new algorithm and successfully used the new features of the GPU to speed up the computation and to achieve the desired performance. For a resolution of 4896×3286 pixels we achieved a time of about 40 milliseconds. This is well within our goal of 25

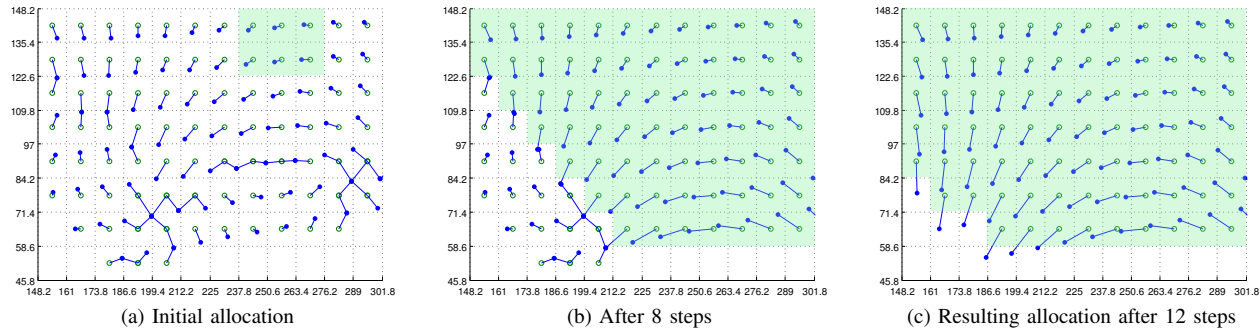


Fig. 5

DOT CORRELATION ALGORITHMS APPLIED ON AN EXAMPLE

Table 1

RUNTIMES AND SPEEDUP FOR DIFFERENT RESOLUTIONS AND DOT NUMBERS ON CPU AND GPU

Resolution	Dots	CPU		GPU		Speedup
		Total (ms)	Correlation (ms)	Total (ms)	Correlation (ms)	
1388 × 1038	69 × 52	24.59	18.57	4.68	2.53	5.23
1600 × 1200	78 × 59	46.92	25.40	6.63	3.15	7.07
2048 × 2048	116 × 116	88.14	61.99	14.35	5.55	6.15
4896 × 3280	242 × 162	332.90	220.90	39.32	19.33	8.46

frames per second, even for a resolution of 16 megapixels. Compared to the standard serial approach, we were able to speed up the dot-exploration by a factor of 175.

The next logical step would be to focus on the post-processing with GPGPUs. In this context, the measured lens needs to be reconstructed. This can be done iteratively or by using a Least-Square Fit algorithm. First simple implementations showed a good performance on GPUs.

References

- [1] M. Yamazaki and G. Xu, "3D reconstruction of glossy surfaces using stereo cameras and projector-display," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference*, 2010.
- [2] M. Hissmann, "Bayesian Estimation for White Light Interferometry," 2005.
- [3] 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," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370 – 1380, 2008, general-Purpose Processing using Graphics Processing Units.
- [4] P. Bakkum and K. Skadron, "Accelerating SQL database operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 94–103. [Online]. Available: <http://doi.acm.org/10.1145/1735688.1735706>
- [5] K. Karimi, N. G. Dickson, and F. Hamze, "High-Performance Physics Simulations Using Multi-Core CPUs and GPGPUs in a Volunteer Computing Context," *CoRR*, vol. abs/1004.0023, 2010.
- [6] S. Gobron, A. Çöltekin, H. Bonafos, and D. Thalmann, "GPGPU computation and visualization of three-dimensional cellular automata," *The Visual Computer*, vol. 27, pp. 67–81, 2011, 10.1007/s00371-010-0515-1. [Online]. Available: <http://dx.doi.org/10.1007/s00371-010-0515-1>
- [7] A. Schäfer and D. Fey, "High Performance Stencil Code Algorithms for GPGPUs," in *Computational Science - ICCS*, ser. Procedia. Elsevier, 2011, p. accepted for publication.
- [8] J. Kong, M. Dimitrov, Y. Yang, J. Liyanage, L. Cao, J. Staples, M. Mantor, and H. Zhou, "Accelerating MATLAB Image Processing Toolbox functions on GPUs," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 75–85. [Online]. Available: <http://doi.acm.org/10.1145/1735688.1735703>
- [9] M. Komann and D. Fey, "Realising emergent image preprocessing tasks in cellular-automaton-alike massively parallel hardware," *IJPEDS*, vol. 22, no. 2, pp. 79–89, 2007.
- [10] L. A. Carvalho, "A simple and effective algorithm for detection of arbitrary Hartmann-Shack patterns," *Journal of Biomedical Informatics*, vol. 37, no. 1, pp. 1 – 9, 2004.
- [11] D. Kirk, "NVIDIA cuda software and gpu parallel computing architecture," in *Proceedings of the 6th international symposium on Memory management*, ser. ISMM '07. New York, NY, USA: ACM, 2007, pp. 103–104. [Online]. Available: <http://doi.acm.org/10.1145/1296907.1296909>
- [12] NVIDIA, "Fermi White Paper," 2009.
- [13] —, "NVIDIA Fermi Tuning Guide 1.3," 2010.
- [14] —, "NVIDIA CUDA Programming Guide 3.1," 2010.
- [15] B. Parhami, *Introduction to Parallel Processing Algorithms and Architectures*. New York, USA: Kluwer Academic Publishers, 2002.
- [16] F. Dehne and K. Yogaratnam, "Exploring the Limits of GPUs With Parallel Graph Algorithms," *CoRR*, vol. abs/1002.4482, 2010.
- [17] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," *SIGARCH Comput. Archit. News*, vol. 38, pp. 451–460, June 2010. [Online]. Available: <http://doi.acm.org/10.1145/1816038.1816021>
- [18] R. Seidler, M. Schmidt, A. Schäfer, and D. Fey, "Comparison of Selected Parallel Path Planning Algorithms on GPGPUs and Multi-Core Processors," in *Proceeding GSTF ADPC 2010*, 2010, pp. A133–A139.

Evaluation of HPC architectures for BRAMS numerical weather model

Eugenio Sper de Almeida^{1,2}, Michael Bauer², and Alvaro Luiz Fazenda³

¹Center for Weather Forecast and Climate Studies, National Institute for Space Research, Cachoeira Paulista, SP, Brazil

²Department of Computer Science, The University of Western Ontario. London, ON, Canada

³Institute of Science and Technology, Federal University of São Paulo, São José dos Campos, SP, Brazil

Abstract - *This paper investigates the performance of a weather forecasting application (Brazilian Regional Atmospheric Modeling System - BRAMS) on a number of selected HPC clusters in order to understand the impact of different architectural configurations on its performance and scalability. We simulated atmosphere conditions over South America for 24 hours ahead with BRAMS, using 100 cores as a starting point (100 cores step). An extra set of executions took place from 10 to 100 cores (10 cores step) to identify more details about BRAMS performance. Results reveal differences in BRAMS performance and its relationship with interconnection (technology and topology). In conclusion, interconnection can limit application performance even with code improvement.*

Keywords: Performance, BRAMS, Numerical Weather Prediction (NWP) model, High Performance Computing (HPC), parallel processing, multi-core architecture.

1 Introduction

Increasing resolution has resulted in improved model simulations and predictions of key atmospheric phenomena [1]. As a result, the execution time of Numerical Weather Prediction (NWP) models increase exponentially as the number of grid points increase in the x, y and z directions [2]. This can lead to delays in the timely delivery of meteorological information, resulting in the actual occurrence of the atmospheric phenomena before it can be predicted.

The configuration of HPC resources are critical in ensuring that sufficient computing and communication resources are available to deliver enough performance for the timely use of NWP models. The exponential improvement in the accuracy of these computational models, however, represents a challenge for many meteorological centers. Consequently, NWP models must be tailored to get the best performance provided by an HPC system.

Recently, Fazenda et. al. [3] identified limitations in BRAMS (Brazilian Regional Atmospheric Modeling System) scalability due to algorithm implementation. They identified

bottlenecks in BRAMS code and developed new solutions, leading to a decrease of BRAMS execution time and a gain of scalability on HPC clusters. In addition, they developed an efficient solution for parallelism scalability, showing performance gains up to 700 cores.

In analyzing the Weather Research and Forecasting (WRF) Model performance, a NWP similar to BRAMS, [4] stated that choosing the right interconnect technology was essential for maximizing HPC system efficiency. Slow interconnects delay data transfers between servers slowing execution of simulations and causing inefficient utilization of computational resources. Their results, using 24 servers each with two AMD Quad-Core processors, identified WRF's communication-sensitive points and demonstrated its dependency on high-speed networks and fast CPU to CPU communication.

According to [5], a communication bottleneck in an HPC cluster may lead to a significant loss of overall performance and so network communication is another key factor that affects application performance on HPC clusters.

Rodrigues et al [6] show the impact of applying a process mapping approach in the BRAMS model, since the communication link speeds on a specific cluster vary with process selection. They developed a method to obtain close to optimal application process placement on cluster cores.

Clusters with Intel EM64T (78.4%) and AMD x86_64 (11.4%) processors dominate the TOP500 list [7], a ranked list of general purpose systems of common use for high end applications. These systems use a number of different interconnection technologies: Gigabit Ethernet (45,6%), Infiniband (42,6%), Myrinet (0.8%) or Quadrics (0.2%). Even though only 0.20% of the HPC systems on the TOP500 list report that their interconnection topology is a fat tree, it is likely that many of them build their systems with this topology using Gigabit Ethernet, Infiniband, Myrinet or Quadrics interconnection technology.

In a fat tree network, processors may be interconnected by a tree structure, in which the processors are at the leaves of the

tree, and the interior nodes are switches. When one moves up the tree from leaves to the root, the links become "fatter" [8]. An advantage of a tree structure is that communication distances are short for local communication patterns. A drawback, however, is that the root and higher-level nodes become bottlenecks for more global communication.

This paper investigates BRAMS performance and scalability on a number of different clusters available within SHARCNET (Shared Hierarchical Academic Research Computing NETwork) [9]. BRAMS is a limited area forecast model that runs on a broad range of computational systems: from mono-processor desktops to clusters with many processors. We evaluate the BRAMS performance on GigaBit Ethernet, Infiniband, Quadrics, and Myrinet networks, as well as in different AMD and Intel dual-core and quad-core architectures. In Section 2 we describe BRAMS and SHARCNET. We describe the experiments in Section 3. Performance results from BRAMS execution on different HPC clusters are presented in Section 4 and conclusions are provided in Section 5.

2 BRAMS and SHARCNET overview

The SHARCNET is a consortium of 17 academic and research organizations in Ontario whose primary mandate is to provide shared high performance computing facilities and associated services to enable forefront computational research.

Clusters are the main SHARCNET resources and basically serve for two categories of computing programming models: those allowing serial (non-parallel) application to take advantage of a clusters parallelism and those with explicit parallelization of a program [10]. SHARCNET clusters have different interconnection networks and types of AMD and Intel architectures, based on dual-core and quad-core processor chips.

The Lightweight Directory Access Protocol (LDAP) controls account management, enabling a researcher to access to any of the systems through a single account. On each cluster, the Load Sharing Facility (LSF) performs job scheduling [11]. As a user account belongs to a global storage system, codes compiled on a user account can be executed on any appropriate SHARCNET cluster.

BRAMS, a version of the RAMS [12][13] tailored to the tropics, has explicit parallelization. The BRAMS/RAMS model is a multipurpose numerical weather model designed to simulate atmospheric circulations, well suitable for HPC clusters. Analysis and boundary conditions from an atmospheric global circulation model are the data input for BRAMS simulation, which is governed by a RAMSIN parameter definition file. It contains all parameterization related to a specific simulation [14].

3 Experiments

"Downscaling" refers to a technique used to achieve detailed regional and local atmospheric data by using either fine spatial-scale numerical atmospheric models (dynamical downscaling), or statistical relationships (statistical downscaling). An Atmospheric Global Circulation Model (AGCM) run is typically the starting point for downscaling. The downscaled high resolution data can also then be inserted into other types of numerical simulation tools such as hydrological, agricultural, and ecological models [15].

Many meteorological centers in Brazil use INPE/CPTEC AGCM outputs as input for their regional area models, consequently providing a more accurate forecast at regional and local scale. This AGCM runs four times a day (00, 06, 12 and 18 UTC) providing numerical weather forecast outputs for 15 days ahead with resolution T162L28 mode; T refers to spectral truncation type (triangular) in zonal wave 62 (resolution of 100x100 km) and L refers to the number of vertical levels (28 levels) [16].

We simulated this downscaling approach (Figure 1), with the BRAMS model, to forecast weather 24 hours ahead in a spatial resolution of 20x20 km over South-America (grid size of 340 by 370 horizontal points). The analysis and boundary conditions for the BRAMS model came from INPE/CPTEC AGCM model outputs from October 23, 2010.

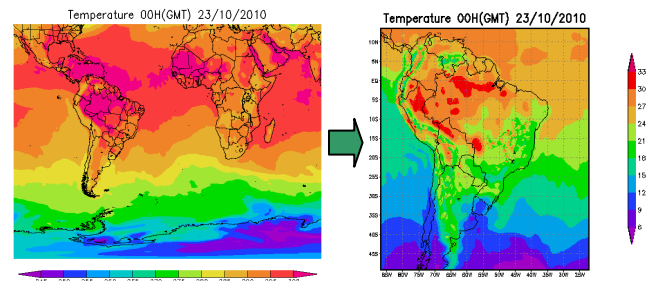


Figure 1. BRAMS downscaling.

NWP models run daily on meteorological centers on HPC resources, at a predetermined window time as part of their operational suite. Scientific visualization tools convert NWP outputs to meteorological maps, meteorologists analyze those maps to produce meteorological forecast and finally publish on meteorological center website for the society.

In order to understand the performance of the BRAMS model, we benchmarked BRAMS execution time starting with 100 cores and incrementing the number of cores by 100. In order to have a closer look at the network influence over the BRAMS performance, we perform additional executions up to 100 processors (incrementing the number of cores by 10). This experiment took place on selected SHARCNET HPC clusters:

- **Bull (384 cores):** HP Linux cluster running XC 3.1 with 96 nodes, four Opteron Mono-Core processor @ 2.4 GHz (QsNet-2/Elan4), and 32 GB of memory;
- **Saw (2688 cores):** HP Linux cluster running XC 4.0 (RHEL 5.1) with 336 nodes, two Xeon Quad-Core processors @ 2.83 GHz (Infiniband), and 16 GB of memory;
- **Requin (1536 cores):** HP Linux cluster running XC 3.1 with 768 nodes, one Opteron Dual-Core processor @ 2.6 GHz (QsNet-2/Elan4), and 8 GB of memory;
- **Narwhal (1068 cores):** HP Linux cluster running XC 3.1 with 267 nodes, two Opteron Dual-Core processor @ 2.2 GHz (Myrinet 2g-gm), and 8 GB of memory;
- **Whale (3072 cores):** HP Linux cluster running XC 3.2.1 with 768 nodes, two Opteron Dual-Core processor @ 2.2 GHz (GigabitEthernet), and 4 GB of memory.

“Bull” and “Narwhal” have direct connected topology interconnects. Fat tree topology interconnects exist on “Saw” (three layers with 2:1 oversubscription), “Requin” (two layers) and “Whale” (three layers) nodes. Table I presents information about switch type, and nominal latency/bandwidth of the selected SHARCNET HPC clusters.

Table I. Latency and bandwidth of SHARCNET clusters interconnection.

Cluster	Interconnection features		
	Switch type	latency (μ s)	bandwidth (MB/s)
Saw	InfiniBand/DDR	1.3	1600
Requin	QsNet2/Elan4	1.4	900
Bull	QsNet2/Elan4	1.4	900
Narwhal	Myrinet 2g (GM)	3.8	250
Whale	GigabitEthernet	50	120

By measuring and comparing BRAMS performance, we extend previous performance analysis from [3]. We compiled BRAMS code using Fortran90/C compilers from Intel and HPMPPI libraries.

4 Performance results and discussions

Message sizes, exchanged by the computing nodes of a HPC cluster, decrease when increasing the number of cores for BRAMS execution.

We identify differences in BRAMS execution time (Figure 2) when increasing the number of cores and we order the HPC clusters according to the best execution time of BRAMS:

- “Requin”, “Bull”, “Narwhal”, and “Saw” up to 60 cores;

- “Requin”, “Saw”, and “Bull” from 70 cores to 80 cores;
- “Saw”, “Requin”, “Bull”, and “Whale” from 90 cores to 200 cores;
- “Saw”, “Bull” “Requin”, and “Whale” for more than 300 cores.

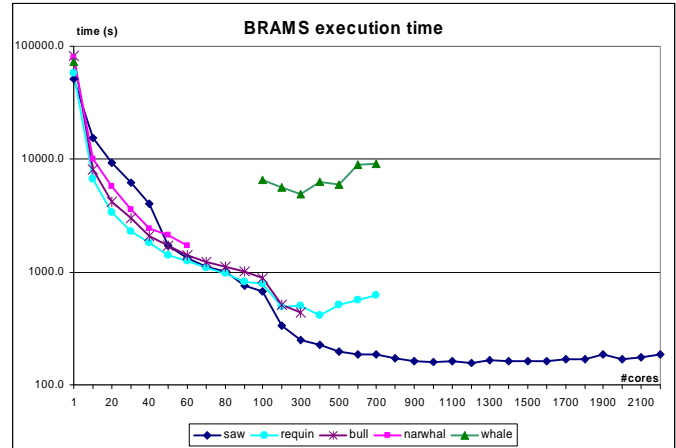


Figure 2. BRAMS model execution time 24h forecast.

BRAMS best execution time was 155.8s on “Saw” with 1200 cores, 410.0s on “Bull” with 376 cores, 416.0s on “Requin” with 400 cores and 1713.4s on “Narwhal” with 60 cores. “Bull” achieved the best BRAMS execution time using its total number of cores (376).

BRAMS execution was limited to 60 cores on “Narwhal” due to unknown problems. “Whale” was decommissioned during our experiment, so we only have results for the 100-700 cores range.

According to Eager [17], speedup and efficiency are the two performance metrics of particular interest when evaluating a parallel system. Speedup (1) is defined as the ratio of the elapsed time when executing a program on a single processor (T_s) to the execution time for n processors ($T_p(n)$):

$$\text{Speedup} = T_s / T_p(n) \quad (1)$$

Efficiency (2) is a metric for the utilization of the n allocated processors. It provides information about how well the processors are utilized in executing a parallel application:

$$\text{Efficiency} = (T_s / (n * T_p(n))) * 100\% \quad (2)$$

Figure 3 shows BRAMS speedup on the systems. The order of the HPC clusters based on the best speedup and efficiency of BRAMS are as follows:

- “Bull”, “Requin”, “Narwhal” and “Saw” up to 60 cores;
- “Bull”, “Requin” and “Saw” from 70 to 90 cores;

- “Bull”, “Saw”, “Requin”, and “Whale” from 100 to 200 cores;
- “Saw”, “Bull”, “Requin”, and “Whale” for more than 300 cores

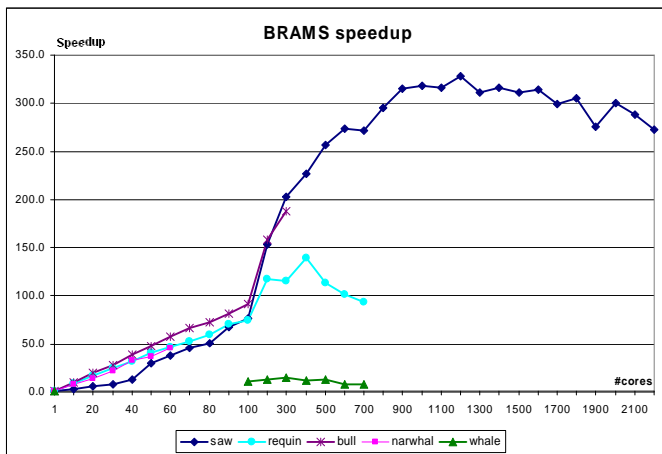


Figure 3. BRAMS model speedup for 24h forecast.

Some of the non-linearity in the execution times and speed-up of BRAMS observed in Figures 2 and 3 arise from multiple latency and bandwidth effects due to system interconnections. Some of these effects can be seen more clearly in the BRAMS efficiency graph (Figure 4).

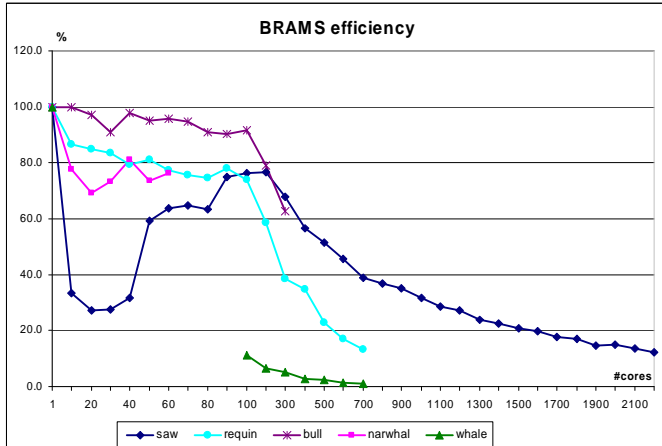


Figure 4. BRAMS model efficiency for 24h forecast.

Our results show that BRAMS performance is not only related to processor performance but that, as demonstrated by [5], switches plays an important role in HPC computing since their latency and throughput increase as packet size grows. Switches with low latencies tend to be more adequate for small message sizes and switches with high bandwidth are more adequate to big message size applications. In other words, applications that exchange small messages take advantage of low latency switches while applications that exchange big messages perform better on high bandwidth switches.

We have identified that the communication processes can become a bottleneck for the scalability of the BRAMS model. Network contention, specifically, is becoming an increasingly important factor affecting overall performance.

In order to gain a deeper understanding of cluster interconnection, we ran Single Transfer Benchmarks (STB) using Intel MPI Benchmarks (IMB) [18] to evaluate cluster MPI latency and bandwidth. It focuses on measuring startup and throughput of a single message transferred between two processes. We used Ping-Pong, where a single message is sent between two processes. Process 1 sends a message of size “x” to process 2 and process 2 sends “x” back to process 1.

Carrying this benchmark between the nearest nodes and farthest nodes of each HPC cluster helped us understand how interconnection affects BRAMS performance. The results from Ping-Pong benchmark revealed that communication with the furthest nodes had a higher latency and lower bandwidth than nodes that were closer. In the worst case, the latency in far nodes increased up to 92.5%, 22.4% and 22.4% in the furthest nodes, respectively for “Saw”, “Requin” and “Narwhal”. In addition, we observed more bandwidth and latency variation between the furthest nodes than in the closest nodes (Figure 5).

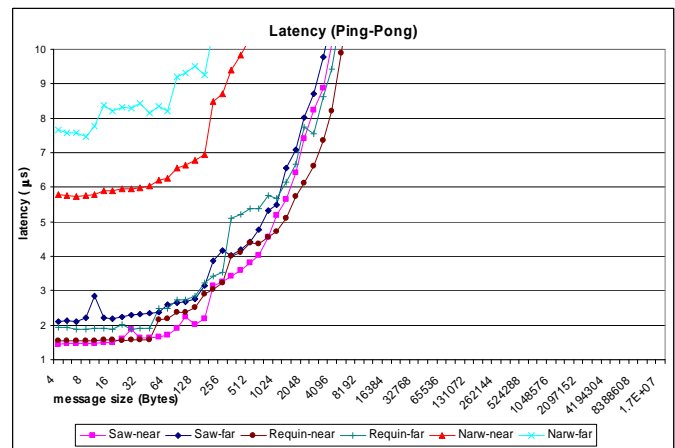


Figure 5. MPI Latency and bandwidth for cluster near and far nodes

We also observed a decrease in the effective bandwidth (Figure 6) to the furthest nodes by 92.6% on “Saw”, 21.4% on “Requin” and 16.8% on “Narwhal”. This was surprising, considering that “Saw” has the interconnection highest bandwidth of them, “Bull” and “Requin” presented a larger effective bandwidth than “Saw” for messages size up to 21 kB.

MPI implementations usually use an eager protocol for small messages and a rendezvous protocol for large messages. The rendezvous protocol needs a handshake between the sender and the receiver, thus requiring host intervention for MPI over InfiniBand and Myrinet. In other words, the rendezvous protocol limits their abilities for overlapping computation and

communication, so messages near critical size do not receive optimal performance. MPI over Quadrics is able to make communication progress asynchronously by taking advantages of the programmable network interface card. Thus it shows much better overlapping potential for large messages [19].

This effect is seen to a greater extent with “Saw” and to a lesser extent with “Narwhal”. Figure 6 shows bandwidth decreases by 33% (794-533 MB/s) on “Saw” and by 17% (178-148 MB/s) on “Narwhal”. This happens for message sizes between 13-21 kB for “Saw” and 16-43 kB for “Narwhal”.

“Bull” presents the same latency and bandwidth values as “Requin” for the closest nodes, since it has the same interconnection technology and all nodes are directly connected to a single switch.

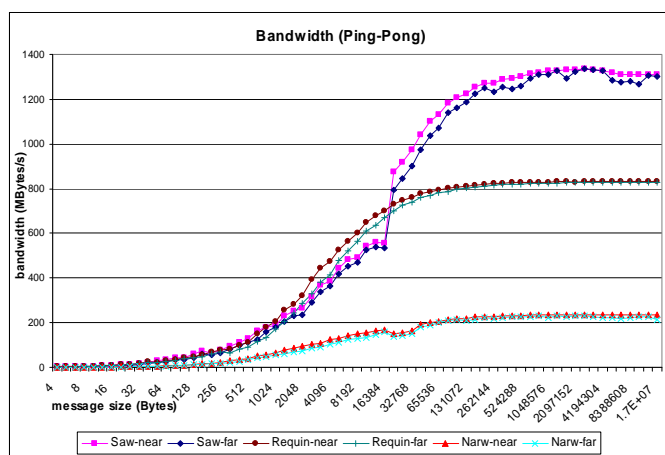


Figure 6. MPI bandwidth for cluster near and far nodes

Despite the nominal switch latency and bandwidth presented in Table I, these values are message size dependent (Figure 5 and 6). So, the higher nominal bandwidth of “Saw” is not reflected in its performance, since this nominal bandwidth is related to a range of message sizes. In reality, latency and bandwidth vary with the size of message exchanged.

The bandwidth decreases by 30% for message sizes smaller than 28K on “Saw”, 7kB on “Bull/Requin”, and 1.7kB on “Narwhal”. The latency increases by 30% for message sizes bigger than 84Bytes on “Saw”, 52Bytes on “Bull/Requin”, and 212 Bytes on “Narwhal”.

We observe that the effective bandwidth for “Saw” is better than “Bull” for message sizes larger than 16KB and for “Requin” for message sizes larger than 21kB. In particular, the effective bandwidth for “Saw” takes a substantial increase for message sizes greater than these. For smaller message sizes, “Bull” and “Requin” have better effective bandwidth

and this leads to better execution time for BRAMS on “Bull” and “Requin” with 70 and 90 cores, respectively (Figure 2).

As can be seen in Figure 6, as the message size increases, so does the effective bandwidth for the systems, though it does so in a non-linear manner. This partially explains the non-linearity of BRAMS performance as the number of cores used grows (Figure 2).

We infer that BRAMS execution time on “Bull”, “Requin” and “Narwhal” is better than on “Saw”, for a small number of cores, mainly because:

- “Saw” has a three layer topology for interconnection, with 2:1 oversubscription, which limits message exchange between nodes;
- The sharp decrease in “Saw” bandwidth for message sizes smaller than 21 kB.

The “Whale” cluster presented the worst performance for BRAMS, mainly because of high latency and low latency of GigaBit Ethernet interconnection.

“Bull” (direct connected topology) has better speedup and efficiency than “Requin” (fat tree topology), though both have the same interconnection (QsNet2/Elan4), because of the interconnection topology. Sometimes on “Requin”, jobs are submitted to nodes connected to the same switch, providing similar performance to that of “Bull”, but at other times jobs are submitted to nodes connected to different switches, increasing the execution time and decreasing BRAMS performance. This happens due to increased latency and lower bandwidth on nodes not connected to the same switch.

BRAMS performance is better in a direct connect topology than in fat tree topology. When using a small number of cores we observe a variation in BRAMS execution due to the effects of the latency introduced by the fat tree connectivity. Despite being the less expensive way to interconnect clusters, it can be difficult to get application performance when compared with direct connect topology [19].

The job submission system allocates cluster nodes according to its scheduling policy and does not consider the interconnection topology. As a result, a job that requires a number of switch ports that match a single switch may require more than one switch in a fat tree topology. For example, we noticed that even with the same interconnection technology, the performance variation is greater in “Requin” than in “Bull”. The fat tree topology of “Requin” requires that a message pass in a certain number of hops for a communication between two cores.

Following the suggestion of Rodrigues et al [6], in this case a process mapping approach could be utilized in order to optimize the overheads between process communications, especially if a fat-tree topology is used. The algorithm used in that paper could be easily adapted to consider a tree structure

representing the different connections linking cores, processor, nodes and switches.

According to [20], choosing a network topology really depends upon the performance you desire, the price you are willing to pay, and perhaps secondarily, the simplicity of the topology and the ability to upgrade the system. In addition, he states that to save costs, typically links are oversubscribed and hence, in practice, we do not see "true" fat tree networks

5 Conclusions

In this paper, we have presented an analysis of BRAMS model performance and scalability over different HPC clusters architectures and configurations of SHARCNET.

As seen from the results obtained from this experiment, even with application code improvement, performance and scalability depends on cluster interconnection technology and topology.

HPC clusters using Infiniband presented the best performance and scalability results for BRAMS, followed by clusters with QsNet, Myrinet and Gigabit Ethernet. However, this order changes with the number of cores involved in BRAMS computation. Clusters with QsNet/Elan4 and Myrinet/2G present better performance for a small number of cores.

We identified how the eager and rendezvous protocols in MPI implementations interfere with interconnection bandwidth performance, especially on HPC clusters with Infiniband, but also with Myrinet, thus affecting application performance.

The results also present the benefits of a direct connect topology over fat tree topology. Even though being a popular topology, a fat interconnection tree represents a challenge in achieving application performance and scalability. Since switch port selection is not part of ordinary job submission system policies, the best performance of an application may not be achieved in a fat tree topology.

6 Acknowledgment

This work was made possible by FAPESP (process: 2010/05823-7) financial support and the facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET:www.sharcnet.ca) and Compute/Calcul Canada. We would like to thank Baolai Ge, from SHARCNET, and Luiz Flávio Rodrigues, from INPE/CPTEC, for their support. The author Alvaro L. Fazenda was partially supported by FAPESP (São Paulo Research Foundation) and CNPq-Brazil (National Council for Scientific and Technological Development).

7 References

- [1] P. J. Roebber, D. M. Schultz, B. A. Colle, and D. J. Stensrud, "Toward improved prediction: High-resolution and ensemble modeling systems in operations", *Weather and Forecasting*, 19, 936–949, 2004. doi: 10.1175/1520-0434(2004)019<0936:TIPHAE>2.0.CO;2
- [2] G. Cats, "24 More Years of Numerical Weather Prediction: A Model Performance model", KNMI Scientific report WR 2008-1, 2008.
- [3] A. L. Fazenda, J. Panetta, P. Navaux, L. F. Rodrigues, D. M. Katsurayama, and L. F. Motta, "Challenges and solutions to improve the scalability of an operational regional meteorological forecasting model", Paper accepted to appear in *Int. J. High Performance Systems Architecture*, 2011.
- [4] G. Shainer, T. Liu, J. Michalakes and J. Liberman, "Weather Research and Forecast (WRF) Model Performance and Profiling Analysis on Advanced Multi-core HPC Clusters", *The 10th LCI International Conference on High-Performance Clustered Computing*. Boulder, CO, 2009.
- [5] B. Huang, M. Bauer and M. Katchabaw, "Hpcbench – a Linux-based network benchmark for high performance networks", *19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*. 2005.
- [6] E. R. Rodrigues, F. L. Madruga, P.O.A. Navaux, J. Panetta, "Multi-core Aware Process Mapping and its Impact on Communication Overhead of Parallel Applications", *Proceedings of the 14th IEEE Symposium on Computers and Communications (ISCC 2009)*, July 2009, doi:10.1109/ISCC.2009.5202271.
- [7] "Top 500 Supercomputer Sites", March 2011. [Online]. Available: <http://www.top500.org/>.
- [8] C. E. Leiserson, "Fat-trees: Universal Networks for Hardware-Efficient Supercomputing, *IEEE Transactions on Computers*, 34(10), October 1985.
- [9] "SHARCNET homepage", March 2011. [Online]. Available: <http://www.sharcnet.ca>.
- [10] C. S. Yeo, R. Buyya, H. Pourreza, R. Eskicioglu, P. Graham and F. Sommers, "Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers", *Handbook of Innovative Computing*, Albert Zomaya (editor), Springer Verlag, 2005.
- [11] M. A. Bauer, "High performance computing: the software challenges", *PASCO '07: Proceedings of the 2007*

international workshop on Parallel symbolic computation. New York, NY, USA: ACM, 2007, pp. 11–12.

[12] “Brazilian Regional Atmospheric Modeling System (BRAMS)”, March 2011. [Online]. Available in <http://www.cptec.inpe.br/brams>.

[13] R. Pielke, W. Cotton, R. Walko, C. Tremback, W. Lyons, L. Grasso, M. Nicholls, M. Moran, D. Wesley, T. Lee, and J. Copeland, “A comprehensive meteorological modeling system – RAMS”, *Meteorology and Atmospheric Physics*, 49(1-4):69–91, 1992.

[14] A. L. Fazenda, D. S. Moreira, E. H. Enari, J. Panetta and L. F. Rodrigues, “First time user's guide (BRAMS version 3.2)”, Cachoeira Paulista. 24p, 2006.

[15] C. L. Castro, R. A. Pielke and G. Leoncini, “Dynamical downscaling: Assessment of value retained and added using the regional atmospheric modeling system (RAMS)”, *J. Geophys. Res.*, 110 , D05108, doi:10.1029/2004JD004721.

[16] J. P. Bonatti, “Modelo de circulação geral atmosférico do CPTEC”, *Climanálise Especial (10 anos Edição Especial)*, 5 pp., 1996, Centro de Previsão de Tempo e Estudos Climáticos (CPTEC), Cachoeira Paulista, Brazil. March 2011. [Online]. Available: www6.cptec.inpe.br/products/climanalise/cliesp10a/bonatti.html.

[17] D. L. Eager, J. Zahorjan, E.D. Lazowska, "Speedup versus efficiency in parallel systems", *Computers, IEEE Transactions on* , vol.38, no.3, pp.408-423, Mar 1989 doi: 10.1109/12.21127

[18] “Intel MPI Benchmarks 3.2.2”, March 2011 [Online]. Available: <http://software.intel.com/en-us/articles/intel-mpi-benchmarks/>

[19] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, D.K. Panda, "Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics", *Supercomputing, 2003 ACM/IEEE Conference* , p. 58, Nov. 2003, doi: 10.1109/SC.2003.10007.

[20] A. Bhatele, “Automating Topology Aware Mapping for Supercomputers”, PhD Thesis, Department of Computer Science, University of Illinois, 2010, <http://hdl.handle.net/2142/16578>.

An Updated Self-stabilizing Algorithm to Maximal 2-packing and A Linear Variation under Synchronous Daemon

Zhengnan Shi

Department of Mathematics and Computer Science, University of Wisconsin, Whitewater, WI, 53190, US
 {shiz@uww.edu}

Abstract—In this paper, we first propose an ID-based, constant space, self-stabilizing algorithm that stabilizes to a maximal 2-packing in an arbitrary graph. Using a graph $G = (V, E)$ to represent the network, a subset $S \subseteq V$ is a 2-packing if $\forall i \in V: |N[i] \cap S| \leq 1$. Self-stabilization is a paradigm such that each node has a local view of the system, in a finite amount of time the system converges to a global setup with desired property. We argue that the algorithm stabilizes in $O(mn)$ moves under any scheduler (such as a distributed daemon). Secondly, we show that the algorithm stabilizes in $O(n^2)$ rounds under a synchronous daemon where every privileged node moves at each round. Thirdly, we propose a variation of the algorithm incorporating a local clock counter on each node. We show that it stabilizes in $O(n)$ rounds under a synchronous daemon.

Keywords: self-stabilization, 2-packing, distributed computing, fault tolerance, graph algorithms

1. Introduction

We define a distributed network as a connected, undirected graph G with node set V and edge set $E \subseteq V \times V$. Let $n = |V|$ and $m = |E|$. Two nodes joined by an edge are said to be *neighbors*. We use $N(i)$ to denote the set of neighbors of node i —its (open) *neighborhood*. $N[i]$, the *closed neighborhood* of i , is defined as $N[i] = N(i) \cup \{i\}$. The *distance* $dist(i, j)$ between two nodes i and j is the number of edge(s) in a shortest i - j path.

A subset S of nodes in a graph $G = (V, E)$ is called a 2-packing [1] if $\forall i \in V: |N[i] \cap S| \leq 1$. A 2-packing is maximal if no proper superset of S is a 2-packing.

The 2-packings are subsets of nodes with some inherent non-locality, in that no two nodes in the 2-packing set can have overlapping neighborhoods. This prevents a node in a 2-packing from having direct knowledge of any other node in the set. A self-stabilizing algorithm for maximal independent sets (1-packing) was produced by Hedetniemi et al. in [2]. Because of the non-locality of 2-packings, our algorithm is more complex both in design, correctness and complexity proofs. The facts below follow directly from the definition of 2-packing.

- 1) Every 2-packing S is an independent set.
- 2) If S is a 2-packing, then $\forall i, j \in S: dist(i, j) \geq 3$.

- 3) If S is maximal 2-packing, then $\forall i \in V \setminus S \exists j \in S: dist(i, j) \leq 2$.

The problem of finding a maximum 2-packing (a maximal 2-packing of largest cardinality) is shown to be NP-hard by Hochbaum and Schmoys in [3], finding a maximal one is easily done in linear time with the standard RAM model.

2. Related Work

Several graph problems arise naturally in distributed systems. For example, distributed algorithms for finding matchings, independent sets, dominating sets and colorings have been studied [4], [5], [2], [6], [7], [8]. Synchronous algorithms have been considered in [9], [10], [11] inter alia. A self-stabilizing algorithm for maximal 2-packing of exponential time is proposed in [12]. Finite complexity k -packing algorithms are proposed in [13]. In [14], using distance- k information, a 2-packing algorithm of $O(n^4)$ moves can be derived under a serial model.

Notably, Gairing et al. [15] presented a self-stabilizing algorithm for maximal 2-packing that requires $O(mn)$ moves using a *Central Daemon*. Using the transformer of Gradinariu et al. [16], it gives a self-stabilizing algorithm for the problem that requires $O(\Delta mn)$ moves using a *Distributed Daemon*, where Δ denotes the maximum degree of the network G .

In this paper, we first propose an ID-based, constant space, self-stabilizing algorithm that stabilizes to a maximal 2-packing in an arbitrary graph. An early version of our algorithm was published [17]. Our algorithm has been changed and its complexity has been significantly improved.

We show that the algorithm stabilizes in $O(mn)$ moves under any scheduler (such as a distributed daemon). The result is an improvement over previous best solution. Secondly, we show that the algorithm stabilizes in $O(n^2)$ rounds under a synchronous daemon where every privileged node moves at each round. Thirdly, we propose a variation of the algorithm incorporating a local clock counter on each node. We show that it stabilizes in $O(n)$ rounds under a synchronous daemon.

3. Our Self-stabilizing Model

Self-stabilization is a strong and desirable fault-tolerance property. The self-stabilizing approach is introduced by

Dijkstra [18]. The contents of a node's local variables are defined as its *local state*. The system's *global state* is the union of all local states. If you take an arbitrary distributed algorithm and start it in a state where its variables have been set to a random value from its domain, the behavior is usually not predictable. However, starting from *any* initial configuration and in *every* execution, self-stabilizing systems are required to recover to a set of legal states.

A self-stabilizing algorithm is presented as a set of rules, each with a boolean predicate and an action. The rules of our algorithms are of the form $p \rightarrow M$, where p is a Boolean predicate, and M is a move which changes local variable(s). A node is said to be *privileged* if the predicate p is true. If a node becomes privileged, it may execute the corresponding move M .

We assume that there exists a *daemon*, an adversarial oracle as introduced in [18], [6], which at each time-step selects one or more of the privileged nodes to move. In the *serial*, also known as the *central daemon* model, no two nodes move at the same time. In the *distributed daemon* model, the daemon can choose any subset of privileged nodes to move simultaneously. A special case of this is the *synchronous daemon* where every privileged node moves at each time-step.

Self-stabilizing algorithms can be designed for networks that are either *ID-based* or for the networks that are *anonymous*. In an ID-based network, each node has a unique ID. In an anonymous network, the nodes lack unique IDs, so there is not a priori way of distinguishing them. It is known that, given IDs, any algorithm for the central daemon can be transformed into one for the distributed daemon (see for example [19]). However, the resulting protocols are not as fast. For a complete discussion of self-stabilization, see the books by Dolev [20] or Tel [21].

When no further state change is possible, we say that the system is in a *stable configuration*. A self-stabilizing algorithm must satisfy:

- 1) From any initial illegitimate state it reaches a legitimate state after a finite number of moves; and
- 2) For any legitimate state and for any move allowed by that state, the next state is a legitimate state.

The complexity of a self-stabilizing algorithm is measured by the upper bound of the number of moves and/or rounds [20], [22]. A *round*, also called a *time-step*, is the minimum period of time where every node that is continually privileged moves at least once. In general, the number of moves is an upper bound on the number of rounds under any daemon.

Our self-stabilizing algorithm uses the shared-variable version of self-stabilizing model. Every node executes the same set of self-stabilizing rules, and maintains and changes its own set of local variables based on the current values of its variables and those of its *neighbors*. Our approach is designed under composite atomicity of communication:

a node is able to read the actual state of its neighbors and update its own state in one atomic step. We assume each node in the network has a unique ID.

4. A Self-stabilizing Algorithm for Maximal 2-packing

We now formally introduce a self-stabilizing algorithm which finds a maximal 2-packing. It is referred to as Algorithm *Maximal 2-Packing*. When we say node i , variable i holds the ID of the node. Local variables of Algorithm *Maximal 2-Packing* are listed next. The algorithm has been changed and optimized from Algorithm 1 in [17].

Algorithm 1: *Maximal 2-Packing*

```

c-Decrease
  if  $c_i = 2 \wedge c_{min-p} = 0$ 
    then  $c_i = 1 \wedge r_i = min-p$ 
Leave
  if  $c_i = 0 \wedge r-p \neq null$ 
    then  $c_i = c_{min-p} + 1 \wedge r_i = rt(min-p)$ 
Join
  if  $min-p = null \wedge c_i \neq 0$ 
    then  $c_i = 0 \wedge r_i = i$ 
c-Orphan
  if  $min-p \neq null \wedge c_i = 1 \wedge (r_i \neq rt(min-p) \vee c_{r_i} \neq 0)$ 
    then  $c_i = c_{min-p} + 1 \wedge r_i = rt(min-p)$ 

```

- 1) Variable c is an enumerate type of 3 possible values: 0, 1 or 2. Variable c is used to record the shortest distance to a node in the 2-packing set S . If $c = 0$, the node is in the set S . If $c = 1$, then the node is adjacent to a node of $c = 0$. If $c = 2$, then the node is adjacent to a node of $c = 1$, and possibly other nodes of $c = 1$ or $c = 2$; but it is not adjacent to any node of $c = 0$. Through neighbors' variable c 's, a node can extract information on whether it is within distance 2 from a node in the 2-packing set S . Subscription is used to denote the ID of the hosting node of a variable. As part of algorithmic design, let $c_{null} = 2 \neq 0$.
- 2) Variables r (root) points to neighbors or its own ID. It is a positive integer designed to hold the ID's of nodes. For nodes of $c = 1$, the variable is actively maintained. For nodes of $c = 0$ or $c = 2$, r is not always maintained depending on the need of Algorithm *Maximal 2-Packing*. Subscription is used to denote the ID of the hosting node of a variable. Note, that variable r may hold a value that is not an ID of any node.

Note that variables c and r do not allow *null* value. Different from Algorithm 1 of [17], we no longer use variable p . The following notations are used to abridge the

presentation of Algorithm *Maximal 2-Packing*. Let i be a node in the network.

- 1) Notation rt is a function which takes the ID of a node as a parameter. $rt(i)$ returns a positive integer. If $c_i = 0$, $rt(i)$ returns i , the ID. If $c_i \geq 1$, $rt(i)$ returns r_i . Note that rt does not allow $null$ value.
- 2) Notation $r-p$ denotes a node of minimum c in set $\{j \in N(i), rt(j) > i \wedge c_j < 2\}$. If there are more than one nodes, $r-p$ is the one with the maximum rt . If there are more than one nodes with the maximum rt , $r-p$ is the one with the maximum ID. If there is no such node, we let $r-p$ be $null$.
- 3) Notation $min-p$ denotes a node of minimum c in set $\{j \in N(i), c_j < 2\}$. If there are more than one nodes, $min-p$ is the one with the maximum rt . If there are more than one nodes with the maximum rt , $min-p$ is the one with the maximum ID. If there is no such node, we let $min-p$ be $null$.

Assume the algorithm runs on node i . The rules of our self-stabilizing algorithm are represented by a list of if-then statements. The rules are tried in the listed order. Upon stabilization, the maximal 2-packing set is the set S of nodes of $c = 0$.

4.1 the Design of Rules

In this section, we describe the reasons behind the design of the rules of Algorithm *Maximal 2-Packing*.

The c-Decrease Rule applies to a node of $c = 2$. If a node i sees a neighbor, $min-p$ with $c_{min-p} = 0$, then i decreases its variable c_i from 2 to 1. The root r_i is also adjusted to the ID of the node $min-p$. After the move, variable c_i reflects the shortest distance from i to a node in set S .

The Leave Rule uses ID's to resolve conflicts when the distance between two nodes in the set S is less than 3. If a node i of $c_i = 0$ is adjacent to a node $r-p$ of $c_{r-p} < 2$ and $i < rt(r-p)$, then i changes its c to a nonzero value. The Leave Rule ensures that the distance between any pair of nodes in S is at least 3. For the Leave Rule to function properly, any node j of $c_j = 1$ must satisfy that r_j equals the largest ID of a neighboring node in the set S . This is guaranteed by the last rule of Algorithm *Maximal 2-Packing*, the c-Orphan Rule.

If node i is surrounded by node(s) of $c = 2$, i considers itself to be of distance 2 from any node in the 2-packing set S . Thus i can move by the Join Rule to join set S . Its root variable is set to the ID of itself. If $min-p \neq null$, the node cannot update its variable r_i . Nevertheless, the variable on a node of $c = 0$ is not used by Algorithm *Maximal 2-Packing*.

A node i of $c_i = 1$ must be adjacent to a node with $c = 0$. The c-Orphan Rule is designed to ensure this property. By definition, notation $min-p$ denotes a node of minimum c in set $\{j \in N(i), c_j < 2\}$. If there are more than one node, $min-p$ is the one with the maximum rt . Hence, the

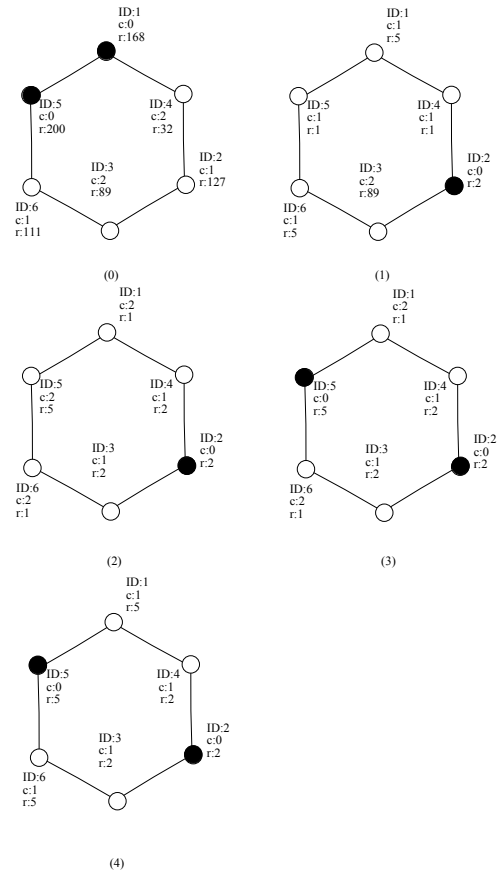


Fig. 1: Network Topology Diagrams of a Sample Execution under Synchronous Daemon

c-Orphan Rule also updates variable r_i to the largest ID possible (without increasing c). This ensures that the Leave Rule functions properly.

5. Sample Executions

In this section, we use examples to demonstrate how Algorithm *Maximal 2-Packing* moves and converges. The adversarial Central Daemon and the Distributed Daemon are nondeterministic. We construct a sample execution of Algorithm *Maximal 2-Packing* under a Synchronous Daemon in Figures 1 and 2. To achieve a longer execution, we introduce a transient error on node 5 which should remain in the set S for the entire execution. However, node 5 misread the ID of node 1 as 7 and it left the set in the first round. The network is a cycle of 6 nodes with ID's 1 through 6.

6. Correctness, Convergence and Complexity Analysis

Despite changes in our algorithm, Lemma 1 of [17] shows the correctness of Algorithm *Maximal 2-Packing*.

Node ID	Local Variables		Time-step 0	Time-step 1	Time-step 2	Time-step 3	Time-step 4
1	c	0	1	2	2	1	5
	r	168	5	1	1	5	
2	c	1	0	0	0	0	0
	r	127	2	2	2	2	2
3	c	2	2	1	1	1	2
	r	89	89	2	2	2	2
4	c	2	1	1	1	1	2
	r	32	1	2	2	2	2
5	c	0	1	2	0	0	5
	r	200	1	5	5	5	5
6	c	1	1	2	2	1	5
	r	111	5	1	1	5	

Fig. 2: Data Table of a Sample Execution under Synchronous Daemon

Lemma 1: Upon stabilization, Algorithm *Maximal 2-Packing* produces a maximal 2-packing set S in which every node has $c = 0$.

We continue to use the concept of a *dirty* node [17]. The complexity of our algorithm is bounded next.

Lemma 2: Sequence the unique ID's of all the nodes in graph G from ID_1 to ID_n . ID_1 is the smallest and ID_n is the largest. Let S_1 be a set of ID's initially empty. We move the nodes in and out of S_1 , but respect a rule. For a node $i \in S_1$ to leave S_1 , another node $j > i$, must join S_1 before i leaves.

Under this construction, a node i of ID_k can join S_1 no more than $n - k + 1$ times; it can leave the set no more than $n - k + 1$ times, $1 \leq k \leq n$.

Proof: Based on the rule, each time that node i of ID_k joins set S_1 grants to increase the cardinality of set S_1 by 1. Node i must leave the set before it can join again. After the first join, each time i leaves set S_1 grants to increase the number of nodes in S_1 with ID's larger than i . Since the total number of nodes whose ID's are larger than i is $n - k$, i can join S_1 no more than $n - k + 1$ times. Hence i can leave no more than the same number of times. ■

Note that node ID_n can enter set S_1 once and it cannot leave the set. Next analysis looks at the number of moves which is an upper bound of the complexity of Algorithm *Maximal 2-Packing* under any scheduler (such as a distributed daemon). Lemma 3 bounds the number of moves by the Join and Leave Rules.

Lemma 3: A node can move by the Join Rule or Leave Rule $O(n)$ times.

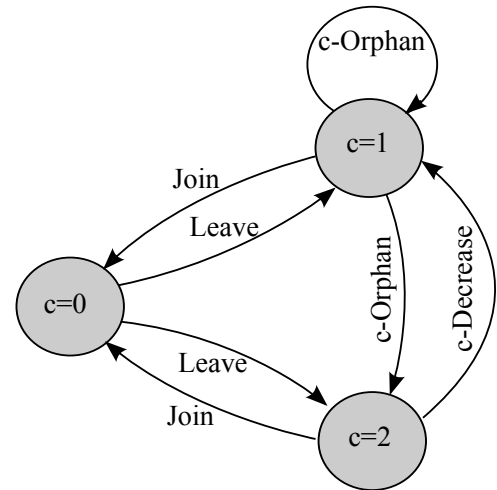


Fig. 3: State Transition Diagram

Proof: Let node i be of ID_k , $1 \leq k \leq n$ of Lemma 2. By the Join Rule, i may apply it only if its $min-p = null$. By the definition of $min-p$, all neighbors of i have $c = 2$. When i later applies the Leave Rule, $r-p \neq null$. By the definition of $r-p$, there are two cases.

If $c_{r-p} = 0$, then $rt(r-p)$ is the ID of node $r-p$. Because $rt(r-p) > i$, the total number of such moves by the Join Rule is no more than $n - k + 1$ by Lemma 2.

For the case $c_{r-p} = 1$: if the node of current $r-p$ was dirty when i moved by the Join Rule, the number of such cases is bounded by the number of dirty nodes $O(n)$. Otherwise, a neighbor of $r-p$, call it j , has moved by the Join Rule. We have $j = rt(r-p) > i$, hence the total number of such move by the Join Rule is no more than $n - k + 1$ by Lemma 2.

If $i \notin S$, one move by the Join Rule allows at most one move by the Leave Rule. Hence the moves by the Join and Leave Rules are of the same order. ■

Next we look at the number of moves by the c-Orphan Rule.

Lemma 4: Let node i have degree δ_i . i can move by the c-Orphan Rule no more than $O(\delta_i n)$ times.

Proof: Except for the first time, any move by the c-Orphan Rule requires a neighboring node of i to move by the Join or Leave Rule. By Lemma 3, each neighbor can move by the Join Rule or Leave Rule $O(n)$ times. Since node i has δ_i neighbors, it can move by the c-Orphan Rule no more than $O(\delta_i n)$ times. ■

Theorem 1: Algorithm *Maximal 2-Packing* stabilizes in a maximal 2-packing set in $O(mn)$ moves under any daemon (scheduler).

Proof: Figure 3 is the state transition diagram with regard to variable c at a node. A move by the c-Decrease Rule, the Join Rule or the Leave Rule changes the value of c . A move by the c-Orphan Rule may result in the same value of c . By Lemma 3, the number of moves by the Join Rule

and leave Rule is of $O(n)$. By Lemma 4, the total number of moves by the c-Orphan Rule is no more than $O(\delta_i n)$. These restrictions bound the total number of moves on all the arcs in the diagram to be of $O(\delta_i n)$. Hence, the total number of moves of node i by Algorithm *Maximal 2-Packing* is $O(\delta_i n)$. Using the notation from Lemma 2, the total number of moves of all the nodes in graph G is:

$$\sum_{i=ID_1}^{ID_n} O(\delta_i n) = O(mn)$$

By Lemma 1, Algorithm *Maximal 2-Packing* produces a maximal 2-packing upon stabilization. ■

We next show Algorithm *Maximal 2-Packing* converges under a synchronous daemon. With a synchronous daemon, every privileged node moves at each *round*. We use *round* to measure complexity for this case. A *round*, also called *time-step*, is the minimum period of time where every node that is continually privileged moves at least once. We cite Lemma 7 of [17] as Lemma 5 follows.

Lemma 5: Under a synchronous daemon, Algorithm *Maximal 2-Packing* can move a constant number of time-steps without any move by the Join Rule before it stabilizes.

Theorem 2: Algorithm *Maximal 2-Packing* stabilizes at a maximal 2-packing in $O(n^2)$ time-steps under a synchronous daemon.

Proof: By Lemma 5, Algorithm *Maximal 2-Packing* can move a constant number of time-steps without any move by the Join Rule. Hence, the time complexity of the algorithm is the same as the number of moves by the Join Rule. Lemma 3 shows that the number of moves by the Join Rule on each node is of $O(n)$. Hence the total number of moves of all the nodes by the Join Rule is of $O(n^2)$. By Lemma 1, Algorithm *Maximal 2-Packing* produces a maximal 2-packing upon stabilization. ■

7. A Variation of the Algorithm with Analysis and Discussion

We propose a variation of the first algorithm *Maximal 2-Packing* for synchronous daemon. Our algorithm incorporates a clock counter algorithm *Synchronous Clock* to manage the execution. Each node in the network has access to a clock ticking variable ct .

We use a natural number t to identify a round. For example, if $t = 1$, time-step t means round 1. We use subscription to denote the 2-packing set S at the end of a round. For example, S_1 is the set at the end of the first round (time-step 1). We use S_0 to denote the set of all the nodes of $c = 0$ before the algorithm starts. At each round, there are nodes joining and leaving the set S . We use J_t to denote the set of nodes joining the set S in round t . Similarly, we

use L_t to denote the set of nodes leaving the set S in round t . For notation, we let $J_0 = S_0$ and $L_0 = \emptyset$. We use a natural number t to identify a round. For example, if $t = 1$, round t means round 1. We use subscription to denote the 2-packing set S at the end of a round. For example, S_1 is the set at the end of the first round (time-step 1). Let the first round as time-step 1.

We introduce algorithm *Synchronous Clock*. It executes round by round together with the second algorithm *SynchronousMaximal 2-Packing*.

- 1) Let variable ct be an unsigned integer. Variable ct is used as a clock ticking counter. On a node i , notation $min-ct$ denotes the smallest value of ct in i 's closed neighborhood $N[i]$, $\{\exists k \in N[i], \forall j \in N[i], ct_k = min-ct \wedge min-ct \leq ct_j\}$.

Algorithm 2 part 1: *Synchronous Clock*

Synchronize Clock

```
if  $ct_i \neq min-ct + 1$ 
then  $ct_i = min-ct + 1$ 
```

The second algorithm *SynchronousMaximal 2-Packing* differs from the first algorithm by the Join Rule, which uses the variable ct in its predicate. A node may join set S when ct is an even number.

Algorithm 2 part 2: *SynchronousMaximal 2-Packing*

c-Decrease

```
if  $c_i = 2 \wedge c_{min-p} = 0$ 
then  $c_i = 1 \wedge r_i = min-p$ 
```

Leave

```
if  $c_i = 0 \wedge r-p \neq null$ 
then  $c_i = c_{min-p} + 1 \wedge r_i = rt(min-p)$ 
```

Join

```
if  $min-p = null \wedge c_i \neq 0 \wedge ct_i$  is even
then  $c_i = 0 \wedge r_i = i$ 
```

c-Orphan

```
if  $min-p \neq null \wedge c_i = 1 \wedge (r_i \neq rt(min-p) \vee c_{r_i} \neq 0)$ 
then  $c_i = c_{min-p} + 1 \wedge r_i = rt(min-p)$ 
```

Lemma 1 is still valid for the correctness of Algorithm 2. We show Algorithm 2 stabilizes in linear time under this setup.

The desired global state of algorithm *Synchronous Clock* is one that each node has the same ct in the entire network and this ct increments by 1 each round.

Lemma 6: Algorithm *Synchronous Clock* stabilizes to its desired global state in $d(G)$ rounds, where $d(G)$ is the diameter of the graph G .

Proof: Let i be a node of minimum ct_i in network G . After the first round, all the node in $N[i]$ have the same

counter value $ct_i + 1$. After $d(G)$ rounds, every node has the same ct in the entire network and this ct increments by 1 each round. ■

Since a distributed daemon is more general than a synchronous daemon, Lemma 5 stands for Algorithm 2.

Theorem 3: In Algorithm 2, *SynchronousMaximal 2-Packing* stabilizes at a maximal 2-packing in linear rounds under a synchronous daemon.

Proof: By Lemma 5, Algorithm *SynchronousMaximal 2-Packing* can move a constant number of time-steps without any move by the Join Rule. Hence, the time complexity of the algorithm is the same as the number of moves by the Join Rule.

Let t be an even nonnegative integer and $J_t \neq \emptyset$. Any node in S_{t-1} are from J_{t-2} or a J set before it. At the end of time-step $t-1$, every node in S_{t-1} is surrounded by neighbors of $c=1$. By the Join Rule, every node in J_t is of distance at least 3 from any node in S_{t-1} . Apply the same argument for J_{t+2} , we have every node in J_{t+2} is of distance at least 3 from any node in S_t . Hence, the node of the largest ID in J_t is guaranteed to stay in set S . In Algorithm 2, the Join Rule requires c_t is even. Every move by the Join Rule increases the cardinality of set S which has at most n nodes. Hence, the algorithm stabilizes in linear time-steps.

The Join Rule relies on algorithm *Synchronous Clock* which stabilizes in linear time by Lemma 6.

By Lemma 1, Algorithm *SynchronousMaximal 2-Packing* produces a maximal 2-packing upon stabilization. ■

8. Concluding Remarks

The problem of 2-packing has inherent non-local properties. However, it is possible to design a reasonably fast self-stabilizing algorithm for it. We first propose an ID-based, constant space, self-stabilizing algorithm that stabilizes to a maximal 2-packing in an arbitrary graph. We show that the algorithm stabilizes in $O(mn)$ moves under any scheduler (such as a distributed daemon). The result is an improvement over previous best solution. Secondly, we show that the algorithm stabilizes in $O(n^2)$ rounds under a synchronous daemon where every privileged node moves at each round. Thirdly, we propose a variation of the algorithm incorporating a local clock counter on each node. We show that it stabilizes in $O(n)$ rounds under a synchronous daemon.

References

- [1] A. Meir and J. W. Moon, "Relations between packing and covering numbers of a tree," *Pacific Journal of Mathematics*, vol. 61, no. 1, pp. 225–233, 1975.
- [2] S. M. Hedetniemi, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani, "Self-stabilizing algorithms for minimal dominating sets and maximal independent sets," *Comput. Math. Appl.*, vol. 46, no. 5-6, pp. 805–811, 2003.
- [3] D. S. Hochbaum and D. B. Shmoys, "A best possible heuristic for the k-center problem," *Mathematics of Operations Research*, vol. 10, no. 2, pp. 180–184, 1985.
- [4] Z. Shi, W. Goddard, and S. T. Hedetniemi, "An anonymous self-stabilizing algorithm for 1-maximal independent set in trees," *Information Processing Letters*, vol. 91, no. 2, pp. 77–83, 2004.
- [5] Z. Shi, "A new self-stabilizing algorithm for maximal 2-packing," in *Proceedings of the International Conference on Computer, Electrical, and Systems Science, and Engineering, (ICCESSE'10)*, vol. 63, March 2010, pp. 327–330.
- [6] S.-C. Hsu and S.-T. Huang, "A self-stabilizing algorithm for maximal matching," *Inform. Process. Lett.*, vol. 43, pp. 77–81, 1992.
- [7] A. Panconesi and A. Srinivasan, "The local nature of δ -coloring and its algorithmic applications," *Combinatorica*, vol. 15, pp. 225–280, 1995.
- [8] S. Rajagopalan and V. Vazirani, "Primal-dual RNC approximation algorithms for set cover and covering integer programs," *SIAM J. Comput.*, vol. 28, pp. 525–540, 1998.
- [9] S. Shukla, D. Rosenkrantz, and S. Ravi, "Observations on self-stabilizing graph algorithms for anonymous networks," in *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995, pp. 7.1–7.15. [Online]. Available: citeseer.ist.psu.edu/shukla95observations.html
- [10] Y. Afek and S. Dolev, "Local stabilizer," *J. Parallel Distrib. Comput.*, vol. 62, no. 5, pp. 745–765, 2002.
- [11] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani, "Self-stabilizing algorithms for orderings and colorings," *International Journal of Foundations of Computer Science*, vol. 16, pp. 19–36, 2005.
- [12] M. Gairing, R. M. Geist, S. T. Hedetniemi, and P. Kristiansen, "A self-stabilizing algorithm for maximal 2-packing," *Nordic J. of Computing*, vol. 11, no. 1, pp. 1–11, 2004.
- [13] F. Manne and M. Mjelde, "A memory efficient self-stabilizing algorithm for maximal k-packing," in *SSS'06: Proceedings of the 8th international conference on Stabilization, safety, and security of distributed systems*. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 428–439.
- [14] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and V. Trevisan, "Distance- k knowledge in self-stabilizing algorithms," *Theor. Comput. Sci.*, vol. 399, no. 1-2, pp. 118–127, 2008.
- [15] M. Gairing, W. Goddard, S. T. Hedetniemi, P. Kristiansen, and A. A. McRae, "Distance-two information in self-stabilizing algorithms," *Parallel Processing Letters*, pp. 387–398, 2004.
- [16] M. Gradinariu and S. Tixeuil, "Conflict managers for self-stabilization without fairness assumption," in *Proceedings of the 27th International Conference on Distributed Computing Systems*, ser. ICDCS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 46–. [Online]. Available: <http://dx.doi.org/10.1109/ICDCS.2007.95>
- [17] Z. Shi, "A linear self-stabilizing algorithm for maximal 2-packing under synchronous daemon with restriction," in *Proceedings of the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, vol. 2, July 2010, pp. 487–493.
- [18] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Comm. ACM*, vol. 17, no. 11, pp. 643–644, Jan. 1974.
- [19] J. Beauquier, A. K. Datta, M. Gradinariu, and F. Magniette, "Self-stabilizing local mutual exclusion and daemon refinement," in *International Symposium on Distributed Computing*, 2000, pp. 223–237. [Online]. Available: citeseer.ist.psu.edu/beauquier02selfstabilizing.html
- [20] S. Dolev, *Self-Stabilization*. MIT Press, 2000.
- [21] G. Tel., *Introduction to Distributed Algorithms, Second Edition*. Cambridge UK: Cambridge University Press, 2000.
- [22] S. Dolev, A. Israeli, and S. Moran, "Uniform dynamic self-stabilizing leader election," *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, no. 4, 1995.

Using OpenCL for Implementing Simple Parallel Graph Algorithms

Michael J. Dinneen, Masoud Khosravani and Andrew Probert

Department of Computer Science, University of Auckland, Auckland, New Zealand

Email: {mjd, masoud}@cs.auckland.ac.nz, apro002@aucklanduni.ac.nz

Abstract—For the typical graph algorithms encountered most frequently in practice (such as those introduced in typical entry-level algorithms courses: graph searching/traversals, shortest paths problems, strongly connected components and minimum spanning trees) we want to consider practical non-sequential platforms such as the emergence of cost effective General-Purpose computation on Graphics Processing Units (GPGPU). In this paper we provide two simple design techniques that allow a non-specialist computer scientist to harness the power of their GPUs as parallel compute devices. These two natural ideas are (a) using a host CPU script to synchronize a distributed view of a graph algorithm where each node of the input graph is associated with a unique processing thread ID and (b) using GPU atomic operations to synchronize a single kernel launch where a set of threads, upper-bounded by at most the number of streaming processing units available, continuously stay active and time-slice the total workload until the algorithm completes. We give concrete comparative implementations of both of these approaches for the simple problem of exploring a graph using breadth-first search. Finally we conclude that OpenCL, in addition to CUDA, is a natural tool for modern graph algorithm designers, especially those who are not experts of GPU hardware architecture, to develop real-world usable graph applications.

Keywords: parallel graph algorithms, GPGPU, OpenCL, CUDA

Contact Author: M.J. Dinneen

Conference: PDPTA'11

I. INTRODUCTION

Parallel programming is a generic concept describing a range of technologies and approaches. However in general it describes a system whereby threads of instruction are executed truly in parallel over a shared or partitioned data source. As part of parallel computing, General Purpose computation on Graphics Processing

Units (GPGPU) is a new and active field. The main goal in GPGPU is to find parallel algorithms capable of processing concurrently huge amounts of data over a number of Graphic Processing Units (GPU). GPGPU involves using the advanced parallel Graphics Processing Unit devices now readily available for general purpose parallel programming. Within GPGPU research, implementing graph algorithms is an important sub-field and is the focus of this paper.

Recently, GPUs have found their places among general computing devices. They are affordable and easily accessible for those enterprises looking for relatively low cost devices to process their massive data. In some applications the size of the input data is so large that even a low-order polynomial-time algorithm surpasses the time limit. Here one may scale down the running time by using more processors to accomplish the computational task concurrently. But then the main challenge is to find a parallel GPU algorithm that accelerates computation with a significant speed up over a well designed sequential one.

CUDA [11] is the GPGPU platform provided by Nvidia Corporation that enables software developers to access the low level instructions and memory of the Nvidia GPUs. With respect to the current architecture of GPUs, CUDA follows the Single Instruction with Multiple Threads approach to parallel processing. While CUDA is restricted to the Nvidia GPUs, OpenCL [8] is a generic overlay with the purpose of providing a common interface for heterogeneous and parallel processing for both CPU and GPU based systems on different devices, such as AMD Radeon graphics cards.

Each GPGPU OpenCL application consists of a host program or script that runs on the CPU and which launches the kernels or kernel programs which are compiled and run on the OpenCL devices. We believe OpenCL makes programming on GPUs easier and safer because it limits access to the kernel (e.g. sandboxing).

Designing parallel algorithms for graph problems has been studied for many years [1], [12], [13]. Implementing these algorithms efficiently on GPUs is a challenging task. In [2], Dehne and Yogaratnam show that one may need to make non-trivial changes to import a PRAM graph algorithm efficiently on GPUs. They mentioned the irregularities among graphs as one of the main challenges. Graph irregularities, as an obstacle in designing fast parallel GPU algorithms for graph problems, is also addressed in [5], [7], and [14].

The Harish and Narayanan paper [6] on parallel GPU algorithms for graphs is widely cited. Another notable result is due to Luo, Wong and Hwu [10]. Both propose parallel implementations of basic graph algorithms which are implemented directly on the Nvidia CUDA platform. As far as we know, the latter paper provides the fastest known breadth-first search graph algorithm for GPUs.

In principle we agree with the authors of [10] that the complexity of a GPGPU algorithm should be the same as the best known sequential one. However, from a practical point of view, simple (possibly non-optimal) correct algorithms are also of value. For instance, when it is known that the expected input cases are relatively small, the extra time and overhead of implementing an optimal algorithm may not be justifiable.

II. TWO DIFFERENT GPGPU DESIGN APPROACHES

We now explain two simple ways that may be used to synchronize graph (and other types of) GPGPU computations where we have a set of well-defined *stages* that need to be completed. For example, in doing a breadth-first search (BFS) in a graph, the stages correspond to the times when the set of nodes at a given depth/level is determined.

A. Host-based synchronization design

The first natural approach is to use a host CPU program (or script) to synchronize stages of a graph algorithm where each part (usually nodes) of the input graph is associated with a unique processing thread. This is a standard way of synchronizing processing threads. Here a global variable, shared by all threads, is set to `false` by the host and set to `true` by any thread inside the kernel that requires another stage.

For example we have the PyOpenCL [9] snippet, shown in Figure 1, from our breadth-first search implementation using host-based synchronization, where `n` is the number of threads.

Note that we use the PyOpenCL method call `enqueue_write_buffer` to send data from the host to the GPU, while `enqueue_read_buffer` will retrieve data from the GPU to the host. Each of the kernel threads will set the global variable `continue_flag` if the algorithm needs another synchronization stage. Note that GPUs operate *asynchronously* from the host. Thus there is a requirement to use an explicit `wait` method call to wait for all kernel tasks to finish before going on to the next stage.

Comment: In its original form the BFS algorithm of [6] uses kernel relaunch to provide a global inter-block barrier between search frontiers. Indeed their program launches two kernels for each iteration—one to check the neighbors for each visited node and another to update the next frontier. In addition to our other changes we have developed an algorithm (see DKP-Host Sync form Section III) that runs in only one kernel launch per stage by using a method for synchronizing threads plus efficiently allocating data among the threads available.

B. Kernel synchronization using atomic operations

The second natural approach is to use GPU atomic operations to synchronize a single kernel launch. In this case a set of threads continuously stays active and time-slices the total workload until the algorithm completes. An important requirement for this approach is having an efficient way to partition an algorithm's workload.

Suppose we have n tasks to complete and we only have a fixed number $m = \text{MAX_THREADS}$ of parallel processing threads. Thus, after evenly distributing, each thread should perform $c = \lceil n/m \rceil$ tasks. This can be done in a number of ways depending on the stride through the tasks t_1, t_2, \dots, t_n . If data is stored in memory we usually want to partition and stride through as $[t_1, \dots, t_c], [t_{c+1}, \dots, t_{2c}], \dots, [t_{(m-1)c+1}, \dots, t_n]$ or as $[t_1, t_{1+m}, t_{1+2m}, \dots, t_{1+(c-1)m}], [t_2, t_{2+m}, \dots, t_{2+(c-1)m}], \dots, [t_m, t_{m+m}, \dots, t_n]$.

The listing of kernel code given in Figure 2 illustrates the distribution of these tasks to processing threads, where `tid` represents one of the active threads operating in parallel. By our convention the thread with `tid=0` does the synchronization management for the algorithm.

In this kernel listing, we use `current_stage` to represent a global clock and each thread keeps a local clock and only executes its set of tasks when they match. Note the use of atomic operations to ensure correctness of shared data.

```

while continue_flag[0]:
    continue_flag[0] = 0
    cl.enqueue_write_buffer(queue, continue_flag_buf, continue_flag)
    cl.enqueue_write_buffer(queue, current_stage_buf, current_stage)
    cl.enqueue_nd_range_kernel(queue, kernel, (n, 1), None)
    cl.enqueue_read_buffer(queue, continue_flag_buf, continue_flag).wait()
    current_stage[0] += 1

```

Fig. 1. Host-based synchronization using PyOpenCL.

```

while (1) // spin lock
{
    // using current_stage as global clock
    if (*current_stage == local_clock[tid])
    {
        // Is everything done?
        if (continue_flag[*current_stage] == 0) return;

        // process n/MAX_THREADS work at this sync time stage
        // ...
        //if needed, we set next continue_flag[*curent_stage+1]=1

        atom_inc(finish_count); // this work thread is done
        local_clock[tid] += 1;
    }

    if (tid==0) // thread 0 detects if everybody is done with stage
    {
        while (atom_cmpxchg(finish_count, MAX_THREADS, 0) != 0) {
            atom_inc(current_stage);
        }
    }
} // end kernel's algorithm loop

```

Fig. 2. Thread-based synchronization in OpenCL kernel.

The `finish_count` value is used by this kernel to synchronize the threads or processors involved in between stages. For example our Nvidia C2050 device has `MAX_THREADS=1024` and we used the second stride technique, described above, in our program DKP-Kernel Sync, which is discussed in Section III.

Comment: Initially we experimented with a lock based inter-block barrier as described in the paper of Xiao and Feng [15]. This has worked reliably for small graphs and for very dense graphs. Unfortunately, Nvidia Corporation do not officially support this inter-block barrier technique and its use can lead to unpredictable results when run on the current family of Nvidia GPGPU offerings. We

eventually came up with our own single-block synchronization technique (presented above) that makes use of atomic operations without the disadvantages of the approach of [15].

III. EXPERIMENTAL RESULTS

As an illustrative example we develop two BFS algorithms similar to the ideas first presented as CUDA implementations by Harish and Narayanan [6]. We have recompiled it to run on our platform (see below) to get comparable running times. Partly due to the available precession of timing GPU computations, all our result times are given in milliseconds elapsed. Calculation times are the kernel run-time from launch until after the host program's final `wait` call. As with common

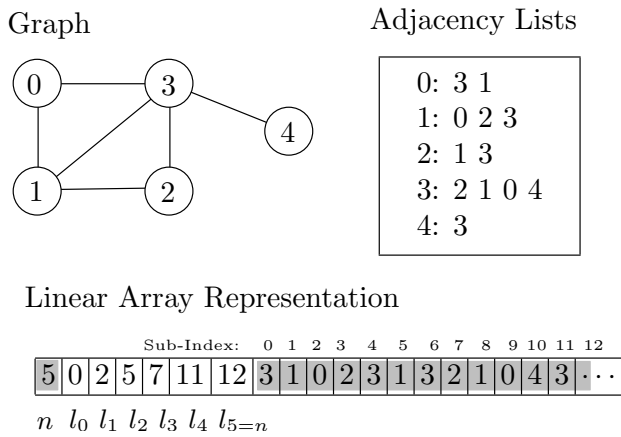


Fig. 3. An effective way to represent sparse graphs in an array.

practice we do not include disk I/O time and host to device copy times. We argue that often an application will copy a graph to memory or GPU and run many algorithms upon that copy (often set 'read-only'). So the real issue is how fast, assuming the graph data structure is available, does the actual algorithm take.

For graph algorithms (on sparse graphs) one often prefers adjacency lists over adjacency matrices since it is easier to iterate through the neighbors of a node in time proportional to the node's out-degree [3]. For a GPU representation one usually linearizes this two dimensional adjacency lists to a one dimensional array such that no loss of efficiency occurs. We represent a "flattened adjacency list" representation of a graph of n nodes and m edges as a vector of length $O(n + m)$ consisting of $[n, l_0, \dots, l_n, v_0, v_1, \dots, v_{m-1}]$. Here n is the order, l_i is the index in this vector of the first neighbor of node i (i.e., points to some v_j index, $0 \leq j < m$). Figure 3 illustrates this array representation. In particular, l_n (plus the sub-index offset $n + 2$) is an index one past the end of the vector and the degree of node i is $l_{i+1} - l_i$.

The expected performance of all three tested algorithms, mentioned in Table I, is $O(nx + m)$, for a graph of order n , eccentricity x (distance of a farthest node from the starting node) and size m . We note that for sparse graphs the value of x is much less than $\log n$, based on the known average height of a random rooted tree. So the dominant term is m in the complexity of these algorithms for most graphs—thus, those chosen as our tests cases are exceptions.

There are a number of variants of the BFS algorithm. One can gather information about predecessors or parents, about BFS "tree" levels and about the list of children. In addition one can gather all the BFS trees for

each of the possible starting nodes, which approaches the task of computing the distance matrix.

A. Test cases and benchmarking environment

As a selection of somewhat "tough" test cases as suggested by Luo, Wong and Hwu's paper [10], we picked sparse graphs from the 9th DIMACS Implementation Challenge [4]. We took each of the 51 state road [di]graphs and made them connected so a BFS from starting node 0 would span and process the entire graph. We removed loops and connected the graphs by adding $k - 1$ arcs to connect those with $k > 1$ components; e.g. for each lowest node index i not in the first component, we added arc $(i - 1, i)$ to the graph. The orders (number of nodes), sizes (number of arcs), and eccentricity of node 0 (distance of farthest node from node 0) are listed in the first few columns of Table I.

The system used by the authors in implementing the BFS graph algorithm, using the above two design approaches, consists of a rack-mountable server with 2 quad core 2.5GHz Intel CPUs and 2 Nvidia Tesla C2050 series (Fermi class) cards. The Tesla C2050 is classified as having Nvidia compute capability 2.0 which defines a range of attributes. In particular, the C2050 has 14 multiple processors (MPs) each with with 32 cores and 3Gb cache (global memory). Each of the 448 cores operate at a frequency of 1.15 GHz. The Tesla C2050 supports block sizes of up to 1024, which can be viewed as the MAX_THREAD value discussed earlier.

B. Observations from our experiments

All programs produced the same (correct) expected BFS depths for each node as a standard CPU-based BFS program. In addition to computing depths from the source node that the original Harish-Narayanan algorithm computes, we also record in our DKP programs, a BFS parent and (later) performed a sequential BFS dag search to verify correctness by ensuring that each parent is, in fact, adjacent and has depth one less than the child.

We conclude that both the OpenCL and CUDA approaches have very little difference in overall performance. Here, in addition to what is reported, we actually converted our OpenCL DKP-Kernel Sync implementation to a pure CUDA implementation and measured the running times on these same test cases. The times, in all cases, were roughly $\pm 2\%$ of those times that are listed in the last column of Table I.

There are a couple of extreme cases (MI and MO) in our experiment that we do not fully understand why the times are so high relatively to the other two programs

TABLE I
GPU BFS ALGORITHM RUNNING TIMES (IN MILLISECONDS) ON USA STATE ROAD GRAPHS (ORIGINATING FROM NODE 0).

State Graph	Nodes	Arcs	Eccentricity	Harish-Narayanan	DKP-Host Sync	DKP-Kernel Sync
AK	69082	156628	1065	491	445	60
AL	566843	1312639	860	469	335	346
AR	483175	1114762	918	496	422	316
AZ	545111	1328587	868	479	394	336
CA	1613325	3959363	1398	949	749	1475
CO	448253	1069293	1081	582	512	340
CT	153011	371483	475	220	206	56
DC	9559	29713	135	63	46	8
DE	49109	119906	293	145	219	19
FL	1048506	2636287	1121	688	544	789
GA	738879	1722510	1234	716	571	615
HI	64892	153136	858	425	431	49
IA	390002	984733	661	356	291	193
ID	271450	634031	1207	560	535	237
IL	793336	1997423	901	529	433	495
IN	497458	1239720	822	447	407	292
KS	474015	1188739	841	462	603	285
KY	467967	1045059	1028	556	467	339
LA	413574	992297	891	475	422	266
MA	308401	764479	785	366	362	181
MD	265912	630270	1335	615	597	254
ME	194505	428316	708	327	362	106
MI	673534	1667305	1338	764	4020	604
MN	547028	1322010	829	457	403	322
MO	675407	1597920	1223	692	6358	559
MS	413250	956059	1021	598	473	299
MT	317905	722443	1286	600	479	290
NC	887630	2008217	1692	1011	800	986
ND	210801	514730	737	340	296	121
NE	308157	768295	594	285	264	144
NH	116920	264862	535	268	194	50
NJ	330386	864564	698	369	497	175
NM	467529	1127802	615	335	339	218
NV	261155	621605	904	419	417	177
NY	716215	1782170	1253	716	1879	605
OH	676058	1667574	898	512	537	423
OK	540981	1310114	1102	614	538	413
OR	536236	1249590	1053	578	1580	392
PA	874843	2165156	971	581	464	582
RI	53658	137695	375	171	141	24
SC	463652	1095125	979	531	466	321
SD	212313	510817	750	348	327	125
TN	583484	1340810	1410	780	690	556
TX	2073870	5143333	1609	1181	973	2158
UT	248730	590269	817	377	381	156
VA	630639	1419559	2172	1222	1347	897
VT	97975	214489	536	248	290	44
WA	575860	1344980	1187	664	797	468
WI	519157	1253811	595	327	358	230
WV	300146	657399	954	442	396	212
WY	253077	607651	747	385	372	146
Total				26232	35457	18753
Average				514	695	368

(we reran each program a few times to double check the reliability of our times). Recall the Harish-Narayanan also uses host synchronization but the program is purely in C, not PyOpenCL—but we do not believe that difference is the cause.

It turns out that for large graphs such as the USA road graph with 24 million nodes, DKP-Host Sync (about 20 seconds GPU time) is about 3–5 times faster than DKP-Kernel Sync. It appears that at about one million nodes (on sparse graphs) the DKP-Host Sync runs faster than DKP-Kernel Sync (e.g. see the big states such as CA, FL and TX). However for these very rare extreme cases, it might be better to use a more optimal algorithm such as the one given in [10]. For small (or dense) graphs one should probably prefer the DKP-Kernel Sync program.

Finally we want to mention that the common “best practices” of ensuring memory coalescence should not necessarily be taken as absolute advice. Our DKP-Kernel Sync program actually performs better with memory strides of increments of MAX_THREADS compared to a version that does memory strides of distance one. We suggest the user try several equivalent variations of their program and take the best performer targeted for their expected input cases.

IV. CONCLUSIONS AND OPEN PROBLEMS

In this paper we introduced and compared two simple techniques for synchronizing processes in parallel graph algorithms. We first considered the scenario where the host CPU is responsible for the synchronization of kernel launches. For example, in DKP-Host Sync, each node of the graph is uniquely associated with a multiprocessor thread. For our second approach we consider the case where the input is partitioned into at most the number of possible parallel threads. For example, in DKP-Kernel Sync, we use only one block (work group) of parallel threads to avoid inter-block synchronization issues. Here the multiprocessor thread use atomic operations for synchronizing the computation. Our experiments showed that both these approaches work well for specific categories of graphs. DKP-Kernel Sync is better for small and dense graphs, while DKP-Host Sync is more efficient on sparse graphs of over a million nodes.

We also compared the running times among the different implementations of the same algorithm via OpenCL and CUDA. We noticed that there is no remarkable difference in computation time between them. Hence OpenCL seems to be as mature and usable as CUDA, with at least one additional advantage of being portable onto more devices (CPUs and GPUs).

There are many problems left to be investigated in this area. For example, we are interested in testing other graph algorithms via these synchronization techniques. Also finding a way to reliably implement an inter-block barrier on the GPU platforms would be extremely valuable. In addition, further work could include developing a OpenCL library of efficient parallel graph algorithms for GPUs.

ACKNOWLEDGMENTS

The authors would like to thank both P.J. Narayanan and Wen-mei Hwu for providing samples of their BFS GPU code for comparison and Radu Nicolescu for discussions and encouragement in designing GPGPU graph algorithms.

REFERENCES

- [1] F. Y. Chin, J. Lam John and I. Chen, Efficient parallel algorithms for some graph problems, *Communication of ACM*, 25(9) 1982, 659–665.
- [2] F. Dehne and K. Yogaratnam, Exploring the Limits of GPUs With Parallel Graph Algorithms, <http://arxiv.org/abs/1002.4482>, 2010.
- [3] M. J. Dinneen, G. Gimel'farb, and M. C. Wilson. *Introduction to Algorithms, Data Structures and Formal Languages, 2nd Edition*. Pearson (Education New Zealand), 2009. ISBN 978-1-4425-1206-1.
- [4] D. Schultes. 9th DIMACS Implementation Challenge, <http://www.dis.uniroma1.it/~challenge9/>; USA state road graphs, <http://www.dis.uniroma1.it/~challenge9/data/tiger/>, October 2005.
- [5] Y. Frishman and A. Tal, Multi-Level Graph Layout on the GPU, *IEEE Transactions on Visualization and Computer Graphics*, 13, 2007, 1310-1319.
- [6] P. Harish and P. J. Narayanan, Accelerating large graph algorithms on the GPU using CUD in *IEEE High Performance Computing*, 2007, LNCS 4873, pp 197–208.
- [7] K.A. Hawick, A. Leist and D.P. Playne, Parallel graph component labelling with GPUs and CUDA, *Parallel Computing*, 36(12), 2010, 655–678.
- [8] Khronos Group. Open Standards for Media Authoring and Acceleration, <http://www.khronos.org/opencl>, 2011.
- [9] A. Klöckner. PyCUDA and PyOpenCL: Even Simpler GPU Programming with Python. Nvidia GPU Technology Conference, (see <http://mathematician.de/software/pyopencl>), 2010
- [10] L. Luo, M. Wong, W-M. Hwu, An Effective GPU Implementation of Breadth-First Search in *Proceedings of the 47th Design Automation Conference* (Anaheim, California, NY, 52–55.
- [11] Nvidia, CUDA. <http://www.nvidia.com/cuda/>
- [12] M. J. Quinn and N. Deo, Parallel Graph Algorithms, *ACM Computing Survey*, 16(3) 1984, 319–348.
- [13] V. Rao and V. Kumar, Parallel depth first search. Part I. Implementation, *International Journal of Parallel Programming*, 16(6) 1984, 479–499.
- [14] J. Soman, K. Kishore, P. J. Narayanan, A fast GPU algorithm for graph connectivity. *IEEE International Symposium on Parallel Distributed Processing*, 2010, 1–8.
- [15] S. Xiao and W. Feng, *Inter-block GPU communication via fast barrier synchronization*, Technical Report TR-09-19, Dept. of Computer Science, Virginia Tech., 2009

Design of a Mutual Situation Awareness Control Protocol between Smart Homes by Using Location Transition Model

Mengqiao Zhang¹, Junbo Wang², Zixue Cheng², Yongping Chen¹, and Lei Jing²

¹ Graduate School of Computer Science and Engineering, University of Aizu, Aizuwakamatsu, Fukushima, Japan

² School of Computer Science and Engineering, University of Aizu, Aizuwakamatsu, Fukushima, Japan

Abstract – *Internet of Things (IoT) is becoming one of the hottest research topics. Smart home, traditionally providing comfortable living conditions or warning services to users, can also be combined with each other to provide mutual situation-awareness service which can enhance interactions between remote families. As many closely related people are separated by distance, we propose a mutual situation-awareness protocol between smart homes by using a location-oriented transition model. Based on this model, the system can detect user's location behaviors, including indoor location transition, room-stay duration, and room-entrance frequency, and finally let remote family be aware of such information. When the user's behaviors don't look normal, warning message will be sent to remote family. We focus on the protocol format and use Event-Condition-Action (ECA) rules to define message sending rules. A prototype system has been built and it can detect user's location behaviors correctly, and control sending messages for awareness properly.*

Keywords: Internet of Things (IoT); Protocol of IoT; Application of IoT; Smart homes; Mutual situation awareness

1 Introduction

Internet of Things (IoT) is becoming a hot topic and research field recently. IoT began with the proposal of Auto-ID-Center MIT in 1999 [1], and then was developed by various institutions and organizations in variety of Countries, e.g. ITU Internet reports 2005 by international telecommunication union [2], A Smarter Planet proposed by IBM [3] Roadmap for the future – Internet of Things in 2020 [4], i-Japan Strategy 2015 toward digital inclusion & innovation [5], the concept of Sensing China [6], and NIT (Network Information Technology) [7] in a report to the USA president and congress.

There are many applications using IoT technologies [8], e.g. smart home/office, smart grid, smart city etc. Smart home, as an important application field of IoT, should not only provide comfortable living condition to users in a single house, but also should be combined as a network platform to provide mutual communication for users.

Nowadays, due to work and study reasons, a large number of closely related people, such as lovers, elderly and their grandchildren are living separately by distance. They

are warring about each other's life situations. Though there are various communication methods such as cell phone, email, and web-chatting, people need to initiate a conversation with contact intention. And these intentional communications still cannot fully fulfill close people's desire to keep strong connection with each other. For example, the elderly often want to know their children's and grandchildren's daily life situations, but perhaps because of busy schedule, their children cannot contact them every day. If there are too many calls, it might cause bothering to everyone. Thus, it's necessary to build a mutual unintended connection way based on the situations of users to enhance interaction for closely related people.

On this topic, there are several researches focusing on the communication between remote family members or close people. In [9] [10] Hitomi Tsujita et al. built some daily appliances with synchronize functions to connect objects of remote close people. In [11], Yoshihiro Ito et.al also developed a system as a daily appliance to keep family relationships for apart living people.

However, above researches mainly focused on the synchronizing objects in different homes, i.e. built some daily appliances embedded with sensors for some special communication functions. About the awareness control protocol, there are few researches about it. To realize the mutual communication, not only the methods for synchronizing the objects are necessary, but also an awareness control protocol is required. We should have a mechanism to decide what kinds of data should be sent, when is appropriate to send the data, and how to display the received data at right time and right way. Hence, a proper awareness control protocol should be built to solve the three problems.

In our previous work [12], a physical spatial mutual situation-aware model has been proposed between smart homes to let users contact with remote family members in an unintended way. We built a system to implement a simple mutual situation awareness case: automatically informing remote family member based on the entrance and exit behaviors of a user, e.g. when the user goes outdoors or comes back indoors. Besides, the movements of indoor location also can be sent to the remote family member.

However, previous system, are only based on location-driven the entrance and exit behaviors, which means location transition information will be sent automatically at once when

user's location changes from one room to another room. Though a case study is given in the previous work, there is a lack of a formal representation of the model and an awareness control mechanism to control message by sending timing and filter unnecessary information. Thus there might be various meaningless messages overabundant since users sometimes may change their indoor location very often for some reasons and the remote family members may be bored, annoyed, or become neglectful to be aware of every behavior of the user.

To this end, we propose a mutual situation-awareness control protocol between smart homes, aiming at enhancing interaction between remote families. And based on our previous work, we also propose a location transition model, to represent indoor location transition. In addition, we record user's staying duration and frequency of entering the rooms, to send the messages based on conditions on those records. By using the model, we can get user's daily indoor location information, transfer the information to remote smart home, and finally let the remote family be aware of the user's location situation.

The rest of the paper is organized as follows. Section 2 gives the model of the proposal. Section 3 presents the design of the method in detail and Section 4 gives an overview of system implements and the experiment. Finally, the paper is concluded in Section 5.

2 Model

For improving the previous system, in this paper, we focus on an awareness control protocol to select meaningful data and send the data at right time. We will not only detect the location transition, but also detect user's room-stay duration and room-entrance frequency, which are more meaningful to users. For instance, if a user stays in a bathroom for very long time or goes to lavatory too many times a day, these behaviors may be abnormal. Letting remote family be aware of these possible abnormal behaviors might be more meaningful. me neglectful to be aware of every behavior of the user.

In Figure 1 we use an example of smart homes to show an image of the system. For simplifying the discussion, we assume that there are only two smart homes, Smart Home A and Smart Home B, in the model. Every smart home has several rooms, e.g. living room, kitchen, bathroom, and so on. The curved lines on the ground are used to represent users' routes of changing the location. Every smart home has a server agent for controlling the communication, LED lights and buzzers for displaying the status of the remote home, and U-tiles sensor network (a sensor network embedded with RFID antenna reader and pressure sensors, to capture the user's location transition behaviors. See [13][14] [15] for detail) on the ground for detecting location and its change of users. We assume that every home has one user in this paper.

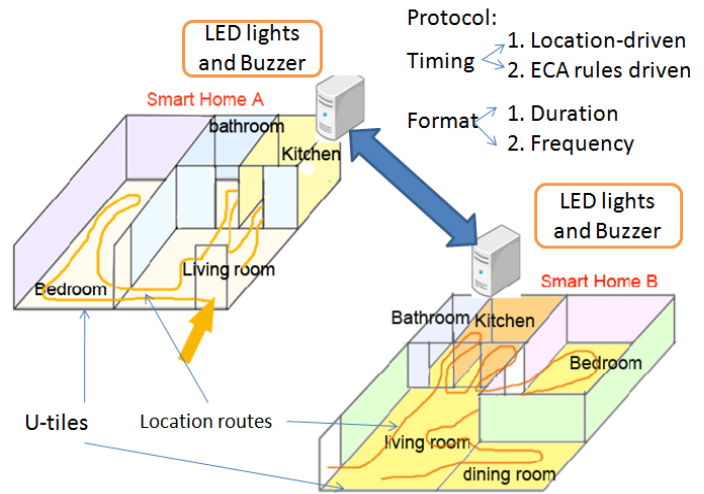


Figure 1. Example of smart homes

We use a directed graph to represent the location transition/change in a smart home. For example, in Figure 2, left side shows a layout of a smart home, and the right side shows a location transition graph which means the user in the home changes location of rooms. In Figure 2, L, Ktn, Bth, and Brm represent Living room, Kitchen, Bathroom, and Bedroom respectively. Regarding the graph, we give some definitions as follows:

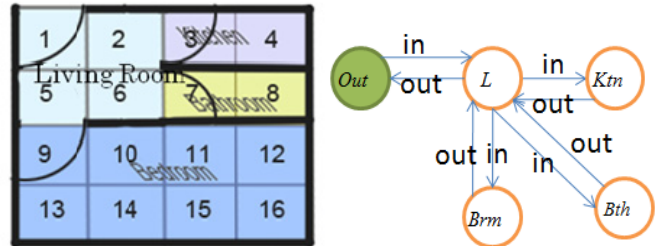


Figure 2. Directed graph for describing location transition

- Node r_i means that the user is in the room r_i . Note that in Figure 2, r_i is presented as L, Ktn, Bth, and Brm.
- Node *Out* means that the user is out of home
- An arrow from node r_i to node r_j means the user can move from room r_i into r_j . Note that the user can move in both directions

What deserves attention is that in this directed graph, we can see node "L" has high in-degree and out-degree, which means Living room (L) is a main passing room into other rooms. We should fully consider that in this space situation, user might walk through it too many times a day without staying. We cannot take this walk-passing behavior as abnormal, if it is reasonable.

3 Design of the system

3.1 The objectives of the awareness control

In the communication part, awareness control protocol needs to solve two problems: sending timing of messages for awareness and the format of the messages. That is, the messages are sent based on not only the entrance behavior but also staying duration and entering frequency, which may reflect the abnormal behaviors. The control is described by Event-Condition-Action (ECA) rules meaning that when a behavior of the user triggers the system, with some conditions being met, some actions will be taken to send the message to the remote home. For the format, we will focus on how to organize the duration and frequency data. Protocol details will be discussed in Section 3.

3.2 Predefined Normal Behaviors and Detected Behaviors

Secondly, for every single room in the smart home, we predefine some ranges or maximum values of normal behaviors including normal room-stay duration and room-entrance frequency. The values are different depending on different room types. Details are shown in the Table 1

3.2.1 Predefined Normal Behaviors (PNB)

$$PNB = \langle r_i, t_i^{\max}, n_i^{\max} \rangle \quad (1)$$

where

- r_i is the ID of a room, where i is an integer to distinguish different rooms.
- t_i^{\max} is the maximum duration time in room r_i (from the user enters room r_i to s/he leaves). The unit of duration time is hour.
- n_i^{\max} is the maximum times that user enters room r_i per day

TABLE I. PREDEFINED NORMAL BEHAVIORS

Room Type	Room ID r_i	Maximum Duration(hour) Per time t_i^{\max}	Maximum Frequency (time) Per day n_i^{\max}
Living room	r_1	t_1^{\max} (e.g. 5 h)	n_1^{\max} (e.g. 5 times)
Kitchen	r_2	t_2^{\max} (e.g. 3 h)	n_2^{\max} (e.g. 4 times)
Bathroom	r_3	t_3^{\max} (e.g. 0.5 h)	n_3^{\max} (e.g. 8 times)
Bedroom	r_4	t_4^{\max} (e.g. 10 h)	n_4^{\max} (e.g. 10 times)

We will decide appropriate values for these parameters firstly, and we will set a bit larger number of frequency if the room is a main passing place. Actually, every user can set the above parameters according to their own life situation.

3.2.2 Detected Behaviors (DB)

In addition to *Predefined Normal Behaviors (PNB)*, user's actual behaviors are needed to be detected and compared to the *PNB*.

$$DB = \langle r_i, t_i^{\text{now}} - t_i^{\text{in}}, n_{ij} \rangle \quad (2)$$

where

- t_i^{in} is the timestamp when user enters room r_i .
- t_i^{now} is the current timestamp when user stays in room r_i .
- n_{ij} is the frequency number that user's j th time to enter room r_i in a day

TABLE II. DETECTED BEHAVIORS

Room Type	Room ID r_i	Detected Duration(hour) Per time $t_i = t_i^{\text{now}} - t_i^{\text{in}}$	Detected Frequency (time) Per day $n_{ij} = n_{i(j-1)} + 1$
Living room	r_1	$t_1 = t_1^{\text{now}} - t_1^{\text{in}}$	$n_{1j} = n_{1(j-1)} + 1$
Kitchen	r_2	$t_2 = t_2^{\text{now}} - t_2^{\text{in}}$	$n_{2j} = n_{2(j-1)} + 1$
Bathroom	r_3	$t_3 = t_3^{\text{now}} - t_3^{\text{in}}$	$n_{3j} = n_{3(j-1)} + 1$
Bedroom	r_4	$t_4 = t_4^{\text{now}} - t_4^{\text{in}}$	$n_{4j} = n_{4(j-1)} + 1$

3.2.3 Detecting Rules

The Table 2 shows how to record user's behaviors. When user enters a room, the system will record the entrance time by plus 1 to the current value of entrance time, supposing the value will be reset once per day. Then we calculate the duration time, if it's longer than normal duration time. The messages will be sent based on those values and conditions on those values, which will be represented by ECA rules discussed in details in 3.4.

3.3 Format of the messages

We define a format to organize the detected data into a message. Table 3 shows the detail fields and lengths of the format of the messages

3.3.1 Fields Definition

- *Room-ID*: Every single room has an ID r_i , e.g. r_1 means a room with ID r_1 .
- *Time*: The time when the message is sent. e.g. 201103082350 means 11:50pm on March 8th, 2011
- *In/Out*: User's status that user is in the room r_i or not. We define that 1 means in; while 0 means out.

TABLE III. PROTOCOL FORMAT

Fields	Room-ID	Time	In/Out	Duration (D)	Frequency (F)	Short Message
Length	16bits	36bits	1 bit	2 bits	2 bits	Optional

- *Duration*: These two bits is to indicate whether and how much the detected duration ($t_i = t_i^{now} - t_i^{in}$) is higher than t_i^{max} . In these 2 bits, the first bit is *high-flag*, and the second bit is *degree-Flag*.

High-flag = 0 means t_i is less than t_i^{max}

High-flag = 1 and *Degree-flag*= 0 means t_i is larger than $t_i^{max} \times 1$ times

High-flag = 1 and *Degree-flag*= 1 means t_i is larger than $t_i^{max} \times 1.5$ times

- *Frequency*: These two bits are to indicate whether and how much the detected frequency n_{ij} is higher than n_i^{max} . In these 2 bits, the first bit is *high-flag*, and the second bit is *degree-flag*.

High-flag = 0 means n_{ij} is less than n_i^{max}

High-flag = 1 and *Degree-flag*= 0 means n_{ij} is larger than $n_i^{max} \times 1$ times

High-flag = 1 and *Degree-flag*= 1 means n_{ij} is larger than $n_i^{max} \times 1.5$ times

- *Short Message*: Some words to describe the situation.

3.3.2 Data Organization Rules

We mainly discuss the fields from *In/Out* to *Short-Message* in details, since they are core parts of the message. Table 4 shows logical rules of value assignment of these fields. It shows valid ways as we mentioned in the fields definition. For example, *Duration* can only be set to 10, 11, or 00. *Frequency* can be in a similar way.

TABLE IV. VALUE ASSIGNMENT

Fields	In/Out	Short Message
Value	1	I'm in XXX
	0	--
Fields	Duration	Short Message
Value	10	Duration exceeds the normal value.
	11	Duration exceeds the normal value too much
	00	--
Fields	Frequency	Short Message
Value	10	Frequency exceeds the normal value.
	11	Frequency exceeds the normal value too much
	00	--

And we take the Short Message field as an optional field, i.e. it can be blank, or can be combined with other short

messages according to different values, e.g. if we set the values as in Table 5, we will get the short message showed in the table.

TABLE V. DATA ORGANIZING CASE

In/Out	Duration (D)	Frequency(F)	Short Message
1	10	11	I'm in XXX. Duration exceeds the normal value. Frequency exceeds the normal value too much

3.4 Message Sending Rules

We use Event-Condition-Action (ECA) method to describe and define the message sending rules in order to control the degree of awareness messages.

Event-condition-action (ECA) rules take the following form:

ON *event*

IF *condition*

DO *actions*

So we can describe the sending rules as follows:

1. Rule 1 (for entering a room)

ON: User enters room r_i

IF: Unconditional

DO: Send message as in/out=1

2. Rule 2 (For the times of enterinng a room)

ON: User enters room r_i

IF: $n_i^{max} < n_{ij} \leq 1.5 \times n_i^{max}$

DO: Send message as in/out=1, F=10

3. Rule 3 (For the times of enterinng a room)

ON: User enters room r_i

IF: $1.5 \times n_i^{max} < n_{ij}$

DO: Send message as in/out=1, F=11

4. Rule 4(for the duration staying in a room)

ON: t_i changes

IF: $t_i^{max} < t_i \leq 1.5 \times t_i^{max}$

DO: Send message as in/out=1, D=010

5. Rule 5 (for the duration staying in a room)

ON: t_i changes

IF: $1.5 \times t_i^{max} < t_i$

DO: Send message as in/out=1, D=011

6. Rule 6 (for existing a room)

ON: User exits room r_i

IF: Unconditional

DO: Send message as in/out=0

3.5 Notification Rules

In order to let remote family be aware of user's location behaviors, we choose to use LED lights and buzzers for display. Firstly, every single room has 2 LED lights, one is for showing remote user's location, and another is for displaying remote user's room-entrance frequency. And we use a buzzer in every single room to notify remote user the duration information. Because when a user stays in a room for an abnormal-long time, a buzzer is a better way to warn remote family than LED light. Especially for the elderly people, remote family should pay more attention to the elderly people's room-stay duration. For instance, if an elderly stays in bathroom for a long time, buzzer warning can attract remote family's attention more easily.

TABLE VI. NOTIFICATION RULES

LED lights Values	In/Out LED	Duration Buzzer	Frequency LED
In/Out=1	On	--	--
In/Out=0	Off	--	--
D=10	--	Buzz one time	--
D=11	--	Buzz 2 times/sec	--
D=00	--	Off	--
F=10	--	--	On
F=11	--	--	Flick 2 times/sec
F=00	--	--	Off

In remote smart home, we define the LED lights and buzzer notification rules when remote family receives the data. According to different values of the fields in a received message, the LED lights and buzzer will work in different ways. Details are showed in Table 6.

4 Implementation and Experiment

4.1 Software Implementation

The system is implemented as shown in Figure 3. In every smart home, U-tiles sensor network was connected to a server agent and the LED lights and buzzer are connected by a Cute-Box [16] which is an embedded board.

For simplifying discussion, we give the implementation architecture in one-way direction from Smart Home A to Smart Home B in Figure 3. Firstly, in Smart Home A, U-tiles sensor network gets location information of the user and send to Server Agent A through UDP socket. In sever agent, there are 3 main Java threads to receive UDP packets, deal with the data, and send the data to remote family. And when a user enters a room, a timer thread will be started to calculate the duration. Then the formatted data will be sent to Server Agent

B in Smart Home B through UDP sockets. In Smart Home B, there are two main threads for receiving UDP packets and sending interpreted data to Cute-Box through serial port. Cute-Box then controls LEDs and buzzer according to the Notification Rules. Thus the location information is sent to another remote smart home in the following way: U-tiles -> Server Agent -> Internet -> Remote Server Agent -> Cute-Box -> LED lights and the buzzer.

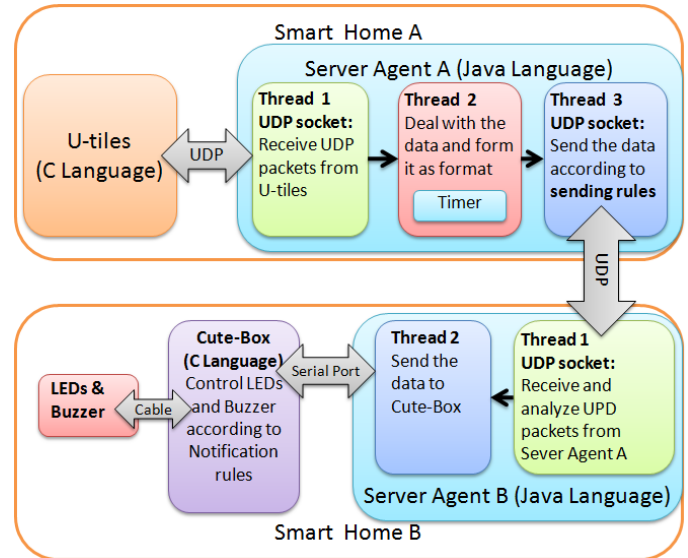


Figure 3. Overview of the system implementation

4.2 Experiment

In Figure 4, two persons were participating the experiment. One was in Smart Home A and another was in Smart Home B. Person A was changing his indoor location on U-tiles sensor network and person B was sitting in another room near the LED lights and the buzzer to confirm the changing of them. For simplifying the discussion, we took kitchen as an instance. We set 0.5h as maximum staying duration time of kitchen and 4 times a day as the maximum frequency. When person A entered the kitchen, the corresponding LED in Smart Home B was turned on. When person A stayed in it for longer than 0.5h, the buzzer buzzed once. And when he continued staying in the kitchen for longer than 0.75h (1.5*0.5h), the buzzer buzzed 2 times per second. When person A entered kitchen for more than 4 times, the frequency LED was turned on. And when person A entered kitchen for more than 6 times (1.5*4 times), the LED flashed 2 times per second. It was confirmed that during the experiment, the LEDs and the buzzer in smart home B correctly reflected the location information of the person A in smart home A.

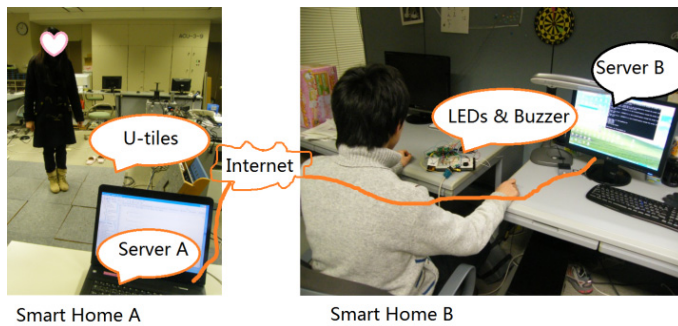


Figure 4. Experiment of the system

5 Conclusion

In this paper, we proposed a mutual situation-awareness protocol between smart homes by using a location-oriented transition model. Based on this model, we can detect user's location transition behaviors including current location, room-stay duration, and room-entrance frequency. We send the location behaviors information to remote family through the protocol, which defines the data organizing format and sending rules by using EAC method. And in remote family smart home, we define displaying rules for interpreting the received data and use LED lights and buzzers to notify user's location behaviors.

We implemented a prototype system and through a basic experiment, we have evaluated the system and it can reflect user's location behaviors correctly. In the future, we will do more detail experiments to evaluate the protocol and the proposed model. And in order to fully realize the mutual situation-awareness service, we plan to improve our protocol design to deal with more complex situations. And more easy-to-use interfaces/devices will be designed to notify users, e.g. wearable devices, smart terminal devices and etc.

6 References

- [1] S. Sarma, D.L. Brock, and K. Ashton, "The Networked Physical World, Proposals for Engineering the Next Generation of Computing, Commerce & Automatic-Identification", Auto-ID Center White Paper, October 2000
- [2] ITU internet report 2005. <http://www.itu.int/pub/S-POL-IR.IT-2005/e>
- [3] "Smart planet". <http://www.ibm.com/smarterplanet/us/en/>
- [4] Internet of Things in 2020, roadmap for the future. <http://www.iot-visitthefuture.eu/index.php?id=30>
- [5] i-Japan. http://www.kantei.go.jp/foreign/policy/it/i-JapanStrategy2015_full.pdf
- [6] "Sensing China". <http://www.ciiot.com/en/Html/index.asp>
- [7] "Report to the president and congress USA". <http://www.whitehouse.gov/sites/default/files/microsites/ostp/pcast-nitrd-report-2010.pdf>
- [8] Luigi Atzori, Antonio Iera, Giacomo Morabito, "The Internet of Things: A survey", The International Journal of Computer and Telecommunications Networking, Volume 54 Issue 15, pp. 2787-2805, Oct 2010.
- [9] Hitomi Tsujita, Koji Tsukada, Itiro Siio, "SyncDecor: Communication Appliances for Couples Separated by Distance", ubicomm, pp.279-286, 2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008
- [10] Hitomi Tsujita, Gregory D. Abowd, "SocialMedicineBox: A Communication System for the Elderly using Medicine Box", In Proceedings of the 12th ACM international Conference Adjunct Papers on Ubiquitous Computing (Ubicomp2010), pp.437-438, Copenhagen, Denmark, September 26 - 29, 2010
- [11] Yoshihiro Itoh, Asami Miyajima, Takumi Watanabe, "TSUNAGARI' Communication: Fostering a Feeling of Connection between Family Member", Proc. of CHI '02 extended abstracts on Human factors in computing systems, New York, USA, 2002
- [12] Mengqiao Zhang, Junbo Wang, Zixue Cheng, Lei Jing, Yongping Chen, et al. "A Remote Mutual Situation-aware Model By Detecting Entrance and Exit Behaviors in Smart Home" Proc. of CICC-ITOE 2011, Vol. 2, pp.65-68, 2011
- [13] J. Wang, M. Kansen, Z. Cheng, T. Ichizawa and T. Ikeda, "Designing an Indoor Situation-Aware Ubiquitous Platform Using Rule-Based Reasoning Method", Proc. of 2008 International Computer Symposium, Taiwan, November 13-15, 2008
- [14] Z. Cheng and T. Huang, "A Situation-aware Support Model and Its Implementation", Proc. of the Japan-China Joint Workshop on Frontier of Computer Science and Technology, pp. 172-177, 2006
- [15] T. Ichizawa, M. Tosa, M. Kansen, A. Yishiyama, Z. Cheng, "The Attribute and Position based Ubiquitous Development Environment Using Antennas with an Automatically Switch", IPSJ SIG Technical Reports, Vol.2006, No.14, ISSN 0919-6072, pp.109-114
- [16] CuteBox wiki: <http://cutebox.wikispaces.com>

A Massively Parallel Algorithm for Polyline Simplification Using an Associative Computing Model

Huy Tran

Department of Computing Sciences
Texas A&M University Corpus Christi
Email: htran1@islander.tamucc.edu

Michael Scherger

Department of Computing Sciences
Texas A&M University Corpus Christi
Email: michael.scherger@tamucc.edu

Abstract - Line simplification is a process of reducing the number of line segments to represent a polyline. This reduction in the number of line segments and vertices can improve the performance of spatial analysis applications. The classic Douglas-Peucker algorithm developed in 1973 has a complexity of $O(mn)$, where n denotes the number of vertices and m the number of line segments. Its enhanced version proposed in 1992 has a complexity of $O(n \log n)$. In this paper, we present a parallel line simplification algorithm using a parallel Multiple-instruction-stream Associative Computing model (MASC). Using one instruction stream of the MASC model, our algorithm has a parallel complexity of $O(n)$ in the worst case using n processing elements.

Keywords: Parallel algorithms, associative computing, SIMD algorithms, line simplification, vertex elimination, level curve

1. Introduction

2D planar level curves are the polylines where mathematical functions take on constant values. In applications such as AutoCAD or MATLAB, these planar level curves are represented as collections of line segments. An example of a level curve in AutoCAD is shown in Figure 1. The number of digitized line segments collected is far more than necessary [2]. Due to the complexity of the geospatial functions, the number of line segments to represent the planar level curve can be very large, which may cause inefficiencies in visual performance. Therefore, the polyline needs to be represented with fewer segments and vertices. It is necessary to perform a polyline simplification algorithm on a 2D planar level curve.

If the line segments were acquired in a stream order, then the end vertex of one line segment is the beginning vertex of the next line segment in the file. It would be straightforward to apply a polyline simplification algorithm. However, in this problem, the line segments of polylines are digitized in a raster scan order. The raster scan ordering of the line segments requires intensive searching on the

remaining set of line segments to reconstruct the polyline ($O(n^2)$ searches).

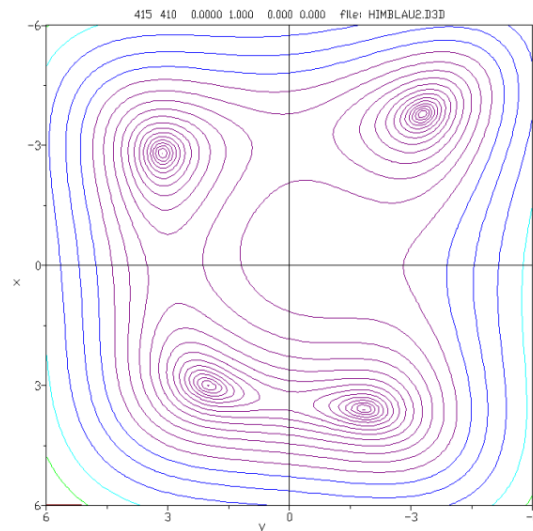


Figure 1: An example of a level curve.

The Douglas-Peucker line simplification algorithm is considered an effective line simplification algorithm [2, 13]. The algorithm uses the *closeness* of a vertex to a segment as a rejection condition. Its worst-case complexity is $O(mn)$, where n denotes the number of vertices and m the number of segments. Furthermore, in 1992 Hershberger and Snoeyink introduced an improvement for Douglas-Peucker algorithm to gain an enhanced $O(n \log n)$ time complexity [4]. The speeding up is achieved by applying binary searches to maintain the path hulls of subchains. Different approaches to this issue have also been discussed in [5, 10, and 12]. However, even the worst-case complexities $O(mn)$ and $O(n \log n)$ are considered computationally expensive when it comes to work with significantly big visualizations.

In this paper, we present a polyline simplification algorithm using the Multiple-instruction-stream Associative Computing model (MASC) [6, 8] to reduce the number of vertices required to represent polylines. MASC is an enhanced SIMD model with associative properties.

By using the constant global operations of the MASC model, our algorithm has a parallel complexity of linear time $O(n)$ in the worst case.

This paper is organized as follows. Section 2 will discuss the polyline simplification in more details and provide a sequential algorithm which our parallel algorithm is based upon. Section 3 will introduce the MASC model of computation and its properties. Section 4 will describe our massively parallel line simplification algorithm using the MASC model. Finally, section 5 will provide the discussions on future work and conclusion.

2. Polyline Simplification and a Sequential Algorithm

Consider the line segments shown in Figure 2. Two connected segments, for example AB and CD, can have one of the four following arrangements:

- A is coincident or near-coincident to C.
- A is coincident or near-coincident to D.
- B is coincident or near-coincident to C.
- B is coincident or near-coincident to D.

The coincidence or collinearity of vertices is determined by investigating their coordinates (x, y). Two near-coincident vertices are considered coincident if their distance is less than or equal to an accepted tolerance α .

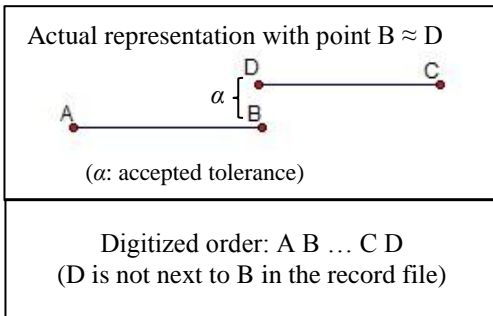
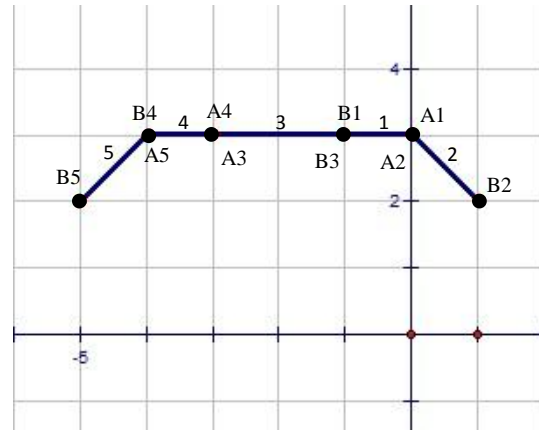


Figure 2: An example of how two connected segments are not in a stream order

2.1. A specific problem

In order to perform the polyline simplification, the digitized line segments in the file need to be re-arranged. The random nature of the digitized line segments necessitates a massive number of search operations to determine coincident points. This is illustrated by the following sequential algorithm to re-arrange these line segments into a stream where line segments having coincident vertices are moved closed to each other. For example,

as shown in Figure 3, five line segments have been digitized. After rearrangement, the stream order would be: B2 A2 A1 B1 B3 A3 A4 B4 A5 B5. Then, each vertex will be checked with its next vertex for coincidence and eliminated accordingly. In this example, points A1, B1, A3, B4 will be eliminated due to coincidence, and points A4, B3 will be deleted due to collinearity.



A1(0, 3)	B1(-1, 3)
A2(0, 3)	B2(1, 2)
A3(-3, 3)	B3(-1, 3)
A4(-3, 3)	B4(-4,3)
A5(-4, 3)	B5(-5, 2)
Segments_Example	

Figure 3: An example of five segments with integer coordinates

2.2. A sequential algorithm

An algorithm to simplify line segments is constructing polylines from coincident and collinear vertices. This can be obtained by eliminating vertices whose distances to the prior initial vertex are less than a maximum accepted tolerance α . The vertices having further distance to the initial vertex ($> \alpha$) could be considered as part of a different polyline. However, finding the coincident and collinear vertices is expensive in this problem.

We call *segArray* the array of line segments. The sequential algorithm is the following:

Begin

1. Set pointer *current* to the first segment in *segArray* (*current=0*)
2. While *current* does not reach the end of *segArray*
 - 2.1. Set pointer *next* to the next segment of *current* segment (*next=current+1*)

- 2.2. While *next* does not reach the end of *segArray*
 - a. Check if the segment in *next* has coincident vertices with *current* segment (the four arrangements discussed in section 2.1)
 - b. If yes, invert *next* segment if needed (in case of the second and forth arrangements in section 2.1)
 - c. Move the *next* segment closed to the *current* segment in the array
 - d. Move pointer *current* to the next segment ($current+=1$)
 - e. Repeat step 2.2
- 2.3. Move pointer *current* to the next segment of *current* segment ($current+=1$)
- 2.4. Repeat step 2

End

The sequential algorithm above is to rearrange line segments into a stream order. The mechanism is similar to the selection sort. The algorithm requires searching all line segments for every investigated line segment to look for the line segment having coincident vertex and move it to the right place. This ineffective searching and sorting can be noticed by the usage of two *while* loops in step 2 and 2.2. Consequently, the complexity of this algorithm is $O(n^2)$, where n is the number of vertices.

3. The MASC model

The following is a description of the Multiple Associative Computing (MASC) model of parallel computation. As shown in Figure 4, the MASC model consists of an array of processor-memory pairs called *cells* and an array of instruction streams.

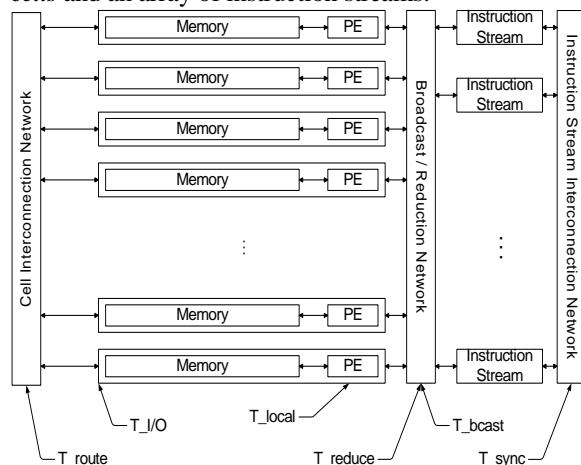


Figure 4: Conceptual view of MASC

A MASC machine with n cells and j instruction streams is denoted as $MASC(n, j)$. It is

expected that the number of instruction stream processors be much less than the number of cells. The model also includes three virtual networks:

1. A cell network used for cell-to-cell communication. This network is used for the parallel movement of data between cells. This network could be a linear array, mesh, hypercube, or a dynamic interconnection network.
2. A broadcast/reduction network used for communication between an instruction stream and a set of cells. This network is also capable of performing common reduction operations.
3. An instruction stream network used for inter-instruction stream communication.

Cells can receive their next set of instructions to execute from the instruction stream broadcast network. Cells can be instructed from their current instruction stream to send and receive messages to other cells in the same partition using some communication pattern via the cell network. Each instruction stream processor is also connected to two interconnection networks. An instruction stream processor broadcasts instructions to the cells using the instruction stream broadcast network. The instruction streams also may need to communicate and may do so using the instruction stream network. Any of these networks may be virtual and be simulated by whatever network is present.

MASC provides one or more instruction streams. Each active instruction stream is assigned to a unique dynamic partition of cells. This allows a task that is being executed in a data parallel fashion to be partitioned into two or more data parallel tasks using control parallelism. The multiple IS's supported by the MASC model allows for greater efficiency, flexibility, and re-configurability than is possible with only one instruction stream. While SIMD architectures can execute data parallel programs very efficiently and normally can obtain near linear speedup, data parallel programs in many applications are not completely data parallel and contain several non-trivial regions where significant branching occurs [3]. In these parallel programming regions, only a subset of traditional SIMD processors can be active at the same time. With MASC, control parallelism can be used to execute these different branches simultaneously. Other MASC properties include:

- The cells of the MASC model consist of a processing element (PE) and local memory. The accumulated memory of the MASC model consists of an array of cells. There is no shared memory between cells.

- Each instruction stream is a processor with a bus or broadcast/reduction network to all cells. Each cell listens to only one instruction stream and initially, all cells listen to the same instruction stream. The cells can switch to another instruction stream in response to commands from the current instruction stream.
- An active cell executes the commands it receives from its instruction stream, while an inactive cell listens to but does not execute the command from its instruction stream. Each instruction stream has the ability to unconditionally activate all cells listening to it.
- Cells without further work are called idle cells and are assigned to a specified instruction stream, which among other tasks manages the idle cells.
- The average time for a cell to send a message through the cell network to another cell is characterized by the parameter t_{route} . Each cell also can read or write a word to an I/O channel. The maximum time for a cell to execute a command is given by the parameter t_{local} . The time to perform a broadcast of either data or instructions is given by the predictability parameter t_{bcast} . The time to perform a reduction operation is given by the predictability parameter t_{reduce} . The time for a cell to perform this I/O transfer is characterized by the parameter $t_{i/o}$. The time to perform instruction stream synchronization is characterized by the parameter t_{synch} .
- An instruction stream can instruct its active cells to perform an associative search in time $t_{bcast} + t_{local} + t_{reduce}$. Successful cells are called *responders*, while unsuccessful cells are called *non-responders*.
- The instruction stream can activate either the set of responders or the set of non-responders. It can also restore the previous set of active cells in $t_{bcast} + t_{local}$ time.
- Each instruction stream has the ability to select an arbitrary responder from the set of active cells in $t_{bcast} + t_{local}$ time.
- An active instruction stream can compute the *OR*, *AND*, *Greatest Lower Bound*, or *Least Upper Bound* of a set of values in all active cells in t_{reduce} time.
- An idle cell can be dynamically allocated to an instruction stream in $t_{synch} + t_{bcast}$ time.

These predictability parameters were identified using an object oriented description of the

MASC model in [11]. They were developed to identify the performance costs using different architecture classes of parallel computing equipment. When the MASC model is implemented using a traditional SIMD computer, it is highly deterministic and the predictability costs can often be calculated and are often “best possible” [7]. Many of the predictability parameters for MASC operations become fixed or operate in one step [7].

4. A MASC Line Simplification Algorithm

Realizing the inefficiency of searching and sorting when coping with the problem as discussed in section 2, we adopt global constant time operations of the MASC model to avoid such inefficiency.

Consider the simple example with five segments having integer-coordinate vertices as shown in Figure 3. In the example, coincident vertices have the same value of coordinates, and three vertices are called collinear if the triangle composed by them has an area value of zero. This can be adjusted in the functions to check coincidence and collinearity by adding an accepted tolerance α .

The input data are described as in Figure 3 as well. Every line of the input file is a line segment consisting of two vertices. Each vertex has an x-coordinate and a y-coordinate. Therefore, at the first state we can determine the vertex's left or right neighbor, which is the other point in the same segment.

We use a tabular organization similar to the one illustrated in Figure 5 as the data structure in our algorithm. That is, the information about left and right neighbors (left\$ and right\$) of the currently investigated vertex and its coincident vertex (coin\$ - if any) are stored in each PE. Vertex A is called on the left of vertex B if A's x-coordinate is less than B's or if A's y-coordinate is less than B's when A and B have the same x value. Vertex A is called on the right of vertex B if A's x-coordinate is greater than B's or if A's y-coordinate is greater than B's when A and B have the same x value. In addition to those location variables, two more variables are defined: visited\$ for tracking if the vertex has been visited and delete\$ for showing if the vertex should be eliminated or not. Furthermore, every vertex is assigned to one PE in the MASC model, which results in a massive number of processing elements.

MASC_LINE_SIMPLIFICATION Algorithm

Begin

1. Set all PEs to active
2. Set del\$ = 'No', visited\$ = 'No'
3. Set left\$/right\$ to the other vertex of the segment
4. For all PEs, repeat until no visited\$ = 'No'
 - 4.1. Find the coincident vertex

- 4.2. If there is no responder (no coincident vertex)
 - 4.2.1. Set visited\$ = 'Yes'
- 4.3. Get the vertex from the responder (if any)
- 4.4. Set empty left\$/right\$ of the two coincident vertices to its coincident vertex
- 4.5. Check if left\$/right\$ of the two coincident vertices and themselves are collinear
 - 4.5.1. If not:
 - a) Set the current PE's del\$ = 'Yes', del\$ = 'No'
 - b) Update field having the deleted vertex as neighbor to its coincident vertex (responders)
 - c) Set visited\$ of both vertices = 'Yes'
 - d) Clear coin\$ of both vertices
 - 4.5.2. Else if they are collinear:
 - a) Set both vertices' del\$ to 'Yes'
 - b) Set the current PE's visited\$ = 'Yes'
 - c) Update fields that have the deleted vertices (responders) as neighbor
 - i. If the deleted vertex is in left\$, update to left\$ of the deleted vertices
 - ii. Else if the deleted vertex is in right\$, update right\$ of the deleted vertices
 - d) Clear coin\$ of both vertices

End

Using the MASC model, our algorithm does not have to re-arrange the line segments because it takes advantage of associative searching. The operations "Find its coincident vertex" in step 4.1 and "Find vertices that have it as neighbor" in step 4.5.1b and 4.5.2c return values in constant time. After the program finishes (all visited\$ are 'Yes'), there would be vertices whose del\$ is 'No'. Those remaining vertices belong to the simplified polylines of the level curve's visual representation. The directions of remaining vertices are maintained with their left\$ and right\$ neighbors.

Figure 5 illustrates the initial table representing the original digitized vertices. During each iteration, a vertex is used in an associative search for its coincident vertex. Then, it checks their neighbors if they are collinear points. Appropriate actions are executed to guarantee that after every round of iteration, there is no deleted vertex in the table, and all vertices will be visited after the program finishes. Figure 6 demonstrates the table after one iteration. The associative searching capabilities of the MASC model helps each round of iteration take constant time. Figure 7 shows the final state of the table after all vertices are visited. Figure 8 is the resultant polyline constructed by fewer segments and vertices. The running time of the

algorithm is $O(n)$ in the worst case when there is no coincidence between vertices.

	vertex	left\$	right\$	coin\$	visited\$	del\$
PE	A1	B1			No	No
PE	B1		A1		No	No
PE	A2		B2		No	No
PE	B2	A2			No	No
PE	A3		B3		No	No
PE	B3	A3			No	No
PE	A4	B4			No	No
PE	B4		A4		No	No
PE	A5	B5			No	No
PE	B5		A5		No	No

Figure 5: The initial table

	vertex	left\$	right\$	coin\$	visited\$	del\$
PE	A1	B1	A2		Yes	Yes
PE	B1		A2		No	No
PE	A2	A2	B2		Yes	No
PE	B2	A2			No	No
PE	A3		B3		No	No
PE	B3	A3			No	No
PE	A4	B4			No	No
PE	B4		A4		No	No
PE	A5	B5			No	No
PE	B5		A5		No	No

Figure 6: The table after one iteration

	vertex	left\$	right\$	coin\$	visited\$	del\$
PE	A1	B5	A2		Yes	Yes
PE	B1	B5	A2		Yes	Yes
PE	A2	A2	B2		Yes	No
PE	B2	A2			Yes	No
PE	A3	B5	A2		Yes	Yes
PE	B3	B5	A2		Yes	Yes
PE	A4	B5	A2		Yes	Yes
PE	B4	A5	A2		Yes	Yes
PE	A5	B5	A2		Yes	No
PE	B5		A5		Yes	No

Figure 7: The tables after all nodes are visited

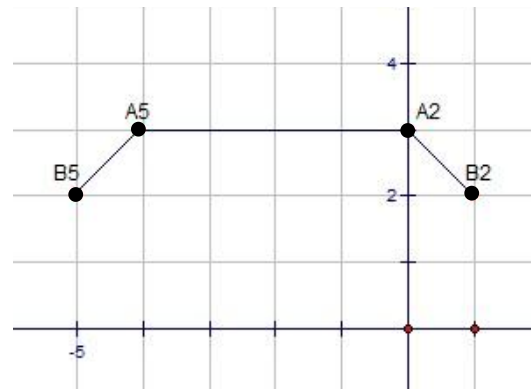


Figure 8: The resultant polyline

5. Conclusion and Future Work

Polyline simplification plays an important factor in visualization applications. The reduction in number of points can help spatial analysis programs improve their performances. The Douglas-Peucker algorithm developed in 1973 is considered an effective solution with the complexity of $O(mn)$ [2], and its enhanced version [4] in 1992 has the complexity of $O(n \log n)$ (m – number of segments, n – number of vertices).

In this paper, a massively parallel polyline simplification algorithm has been introduced. This algorithm has a parallel complexity of $O(n)$ using n processing elements. This speedup is achieved by using a massive number of processing elements and the associative global operations of the Multiple-instruction-stream Associative Computing model (MASC). This algorithm requires only one instruction stream and n processing elements. Therefore, the time complexity linearly depends on the number of vertices. The result of our algorithm as compared to other algorithms is summarized in Figure 9.

Algorithm	Worst-case Complexity	Number of processors	Cost
Sequential	$O(n^2)$	1	$O(n^2)$
Douglas-Peucker	$O(mn)$	1	$O(mn)$
Enhanced Douglas-Peucker	$O(n \log n)$	1	$O(n \log n)$
MASC	$O(n)$	n	$O(n^2)$

Figure 9: Sequential algorithms vs. ASC algorithm

Although this algorithm requires a massive number of processing elements, its parallel computational cost is the same as the Douglas-Peucker algorithm when the number of line segments is high. This research provides an effective solution to this problem.

The future work for this research includes an implementation of the MASC polyline simplification algorithm using the Chapel parallel programming language. Chapel is the choice of language because of its language features in data parallelism, task parallelism, concurrency and nested parallelism via high-level abstractions [1] that closely matches the MASC model. Furthermore, this algorithm will be implemented using CUDA, which is a hardware platform which can also support the MASC model.

The future work of this research also includes an analysis of the average case and best case complexities. The results then can be compared with

the enhanced Douglas-Peucker algorithm complexity [4].

References

- [1] B. L. Chamberlain, D. Callahan, H. P. Zima, "Parallel Programmability and the Chapel Language", Int'l Journal of High Performance Computing Applications, 21(3), pp. 291-312, August 2007.
- [2] D. Douglas, T. Peucker, "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or Its Caricature", Canada Cartographer, Univ. of Toronto Press, 10(2), pp. 112-122, December 1973.
- [3] G. Fox, "What Have We Learnt from Using Real Parallel Machines to Solve Real Problems", Proc. of the 3rd Conf. on Hypercube Concurrent Computers and Applications, vol. 2, ACM Press, pp. 897-955, 1988.
- [4] J. Hershberger, J. Snoeyink, "Speeding Up the Douglas-Peucker line-simplification algorithm", Proc. of the 5th Symp. on Data Handlings, pp. 134-143, 1992.
- [5] G.F. Jenks, "Lines, Computers and Human Frailties", Annuals of the Association of American Geographers, pp. 1-10, 1981.
- [6] M. Jin, J. Baker, "Two Graph Algorithms On an Associative Computing Model", Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques (PDPTA), Las Vegas, June 2007.
- [7] M. Jin, J. Baker, and K. Batcher, "Timing of Associative Operations on the MASC model", Proc. of the 15th IPDPS (Workshop in Massively Parallel Processing), CD-ROM, April 2011.
- [8] J. Potter, "Associative Computing: A Programming Paradigm for Massively Parallel Computers", Plenum Press, New York, NY, 1992.
- [9] J. Potter, J. Baker, S. Scott, A. Bansal, C. Leangsuksun, and C. Asthigiri, "ASC: An Associative-Computing Paradigm", Computer, 27(11), 19-25, 1994.
- [10] K. Reumann, A.P.M Witkam, "Optimizing curve segmentation in computer graphics", Proc. of the Int'l Computing Symposium, pp. 467-472, 1974.
- [11] M. Scherger, "On Using the UML to Describe the MASC Model of Parallel Computation", Proc. of the Int'l Conf. on PDPTA, pp. 2639-2645, Las Vegas, June 2000.
- [12] M. Visvalingam, J.D. Whyatt, "Line Generalization by Repeated Elimination of Points", Cartographic Journal, 30(1):46-52, 1993.
- [13] S.T. Wu, R.G. Marquez, "A Non-self-intersection Douglas-Peucker Algorithm", Computer Graphics and Image Processing, pp. 60-66, Oct 2003.

ViFramework: A framework for networked video streaming components

B. Kersten¹, K. van Rens², and R. Mak¹

¹Security and Embedded Networked Systems, Eindhoven University of Technology, The Netherlands

²ViNotion B.V. The Netherlands

Abstract—*Real-time video content analysis applications for surveillance become more and more demanding. The need for load distribution, remote management and reusability calls for a component framework specialized in networked video streaming applications. Whereas lots of component frameworks exist nowadays, frameworks targeted at networked video streaming are scarce. Added requirements imposed by video surveillance applications include real-time computing and quick failover. The framework proposed in this paper meets these demands by enabling the distributed execution of video streaming applications in an efficient and resource-aware fashion. In this paper, we present the design of the proposed framework and evaluate a prototype implementation. The results of this evaluation show that this implementation is efficient and can successfully perform failover handling, making it suitable for distributing surveillance applications.*

Keywords: Software component framework, video streaming, video surveillance applications, distributed video analysis

1. Introduction

Developing applications using the Component-Based Software Engineering (CBSE) paradigm [1] has many advantages such as high reusability, low time to market and decreased development costs. Many software component frameworks exist nowadays, but frameworks supporting video streaming are rare, especially when network functionality is required.

Applications that need streaming video are Video Content Analysis applications, such as the ones studied in the recent ITEA2 research projects CANTATA [2] and ViCoMo [3]. These applications are becoming increasingly more demanding. Applications that process video streams originating from multiple cameras with computationally-intensive algorithms like object detection and tracking are becoming more common. Due to the high-volume nature of video data, processing components often have high resource demands. The need for distributed applications is motivated by the need for geographical distribution and load distribution in order to make the applications more scalable.

A framework for networked video streaming components is needed, in order to enable component distribution over

hosts connected by a network. The framework must enable components to be configured and composed remotely in order to form an application. By supporting *dynamic reconfiguration* the framework must allow for run-time modifications of the application's component graph.

At design-time, the framework must provide component developers with abstraction of tasks, such as setting up network connections, video compression, timing and multithreading. To make the framework suitable for rapid prototyping, the framework must be flexible and component descriptions easy to adapt. At run-time, the framework should provide network-transparent means to compose an application. In order to allow applications to span both LAN's and WAN's, the framework must support NAT-router and firewall traversal.

Targeting at video surveillance applications, the framework is subject to real-time requirements. In general, a trade-off must be made between timeliness and guaranteed delivery. By adjusting QoS parameters the framework must be able to meet the real-time requirements of the applications. In host-failure situations the framework must be able to perform quick failover, thus increasing robustness and minimizing the amount of lost data.

This paper proposes a framework for networked video streaming components aimed at surveillance applications. An implementation of the proposed framework is presented in [4] and is evaluated in this paper. Evaluation is done by porting an existing surveillance application to the framework after which overhead and failover time is measured.

In Section 2 the general architecture of the proposed framework is presented. An application scenario is sketched in Section 3. Section 4 elaborates on framework details. Framework evaluation is presented in Section 5. Section 6 describes related work and Section 7 concludes the paper.

2. Framework Architecture

The proposed framework exists of a design-time and a run-time part. The design-time part of the proposed framework consists of means that help the component programmer to create components that comply with the framework. This includes an interface definition language, automatic code generation and programming guidelines.

Before existing video content analysis algorithms can be used as components in the framework, they are supple-

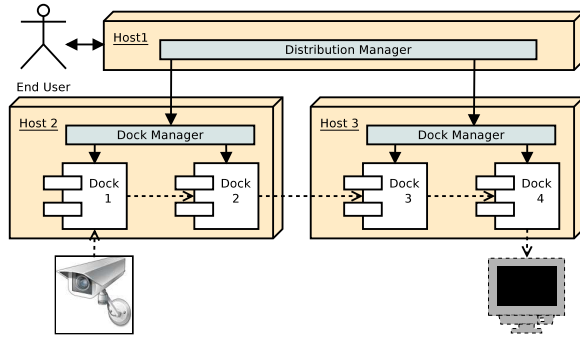


Fig. 1: High-level ViFramework architecture. A video content analysis application consisting of four components spanning two hosts is set up. Each dock manager manages all docks on the host it resides on and one distribution manager controls the whole application.

mented with platform-specific software that provides additional functionality for network usage. This is called the instrumentation procedure and results in a so-called *dock*. The term dock is borrowed from the SOFA framework [5] and denotes a container for multiple components that together provide functionality to the environment. Docks include a control part which controls the included components, handles configuration and facilitates network usage, and a functional part which is the component code.

The run-time part of the framework consists of two active entities that enable the distribution of a video streaming application. Each host that takes part in the framework runs one process that manages all docks on that host. On each host, this *dock manager* is the only process that can instantiate docks. After instantiation the dock manager can configure, start, stop and destroy a dock. Configuration includes binding of the component's interfaces in order to connect them to other components.

All dock managers are connected to a central service named the *distribution manager* that is used to gather information about available hosts from their dock managers. Since the distribution manager has control over all available dock managers, it is capable of composing a networked video streaming application, instantiating and connecting available components. A user-interface to the distribution manager enables end-users of the framework to manually setup an application, although the distribution manager can also be configured to automatically setup and manage pre-defined applications. An overview of the high-level framework architecture is depicted in Fig. 1.

When considering video streaming applications as done in [6], three component types can be distinguished:

- *Input*: Components that capture video data from an input source (e.g. a camera, a file or an Internet stream), convert it to a common internal format, after which it can be offered to an interface.
- *Output*: Components that accept the common internal

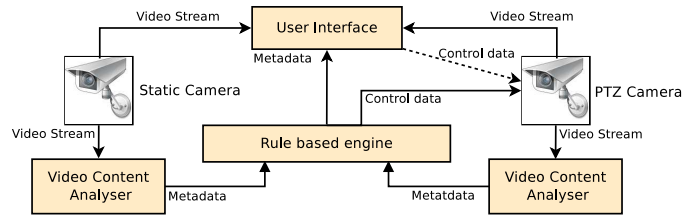


Fig. 2: Data flow diagram of the object detection and tracking application. The control-data flow from the user interface to the PTZ-camera is optional because the end-user may chose to manually control the PTZ-camera or not.

format from an interface and convert it to an output format which can, for example, be a display or a video file.

- *Processing*: Components that can be used to read video data in the common internal format from an incoming interface, process the video stream before forwarding it to an outgoing interface in the same format.

Typically, processing components can reside on any host, whereas in- and output-components need additional hardware in order to fulfill their task and are therefore located in the proximity of these devices (i.e. a camera or video display). If not composed manually, it is the distribution manager's responsibility to setup a pre-defined application taking into account what resources are available on the connected hosts.

The distribution manager is capable of performing failover by re-instantiating failed docks on other hosts and re-routing the data through the re-instantiated docks. In the same way, the distribution manager is capable of performing load distribution. Because video streaming uses a lot of network bandwidth the framework takes network capacity into account when setting up and managing applications. It does so by adjusting stream routes and choosing appropriate Quality of Service (QoS) levels and compression techniques. By using NAT-router and firewall traversal it is possible to deploy applications that cross the borders of a LAN.

3. Application Scenario

As a proof of concept the proposed framework implementation is used to distribute an existing surveillance application over multiple hosts. The application chosen for this is an object-tracking application using a static and a Pan-Tilt-Zoom (PTZ)-camera as depicted in Fig. 2. The video stream from the static camera is used for object detection and tracking. When an object is detected, the PTZ-camera is used to zoom in on the target and to extract more object-specific information. The video stream from the PTZ-camera could, for example, be used for face recognition on the zoomed-in object. The PTZ-camera is controlled automatically using the coordinate information from the "Video Content Analyser" component which analyses the video feed from the static camera. Moreover, the end-user can at any time connect to

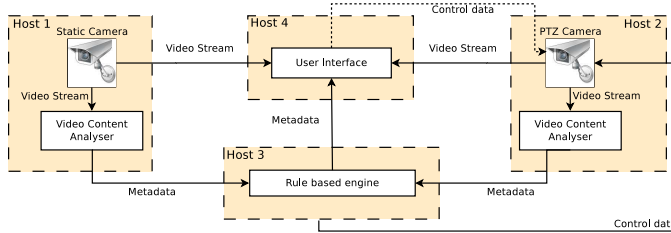


Fig. 3: A possible distribution scenario of the object detection and tracking application spanning four hosts.

the user interface component and watch the incoming video streams and application-generated metadata. Optionally, the user can also take manual control of the PTZ-camera. This application can be divided in up to four docks:

- *Static-camera analysis*: Object detection and tracking algorithms generating PTZ-coordinates based on the video stream provided by the static camera. Metadata describing the objects and their locations is send to a rule-based engine.
- *PTZ-camera analysis*: Controlling the PTZ camera based on incoming PTZ-coordinates and using the video stream from the PTZ-camera as input for a video content analysis algorithm.
- *User interface*: Presentation / interaction component.
- *Rule-based engine*: Gathering metadata from both analysis components and informing the end-user on events by forwarding them to the user interface.

A possible distribution of these docks is depicted in Fig. 3.

4. Framework Details

4.1 Location transparency

The ViFramework provides generic means that allow the end-users to deploy and connect docks on available host irrespective of the underlying network topology. In order to create this *location transparency* the framework is built on top of the XMPP protocol [7] originally designed for messaging purposes. Because of its modularity and ease of extensibility it has become a communication protocol used by all kind of applications such as a the Peer-to-Peer desktop grid computing substrate [8]. XMPP has very attractive features for this framework such as presence information of clients, possibility of NAT-router and firewall traversal and extensive security measures like TLS and SASL. Because of its modularity, a light-weight framework can be created by including only the XMPP modules that are necessary. All this makes the XMPP protocol an excellent network substrate for an easy to extend component-framework that satisfies the needs of demanding video content analysis applications.

A major drawback of the XMPP protocol when used for the ViFramework is the lack of efficient video streaming support. XMPP does have an extension that facilitates stream

initiation (XMPP extension number XEP-0095) that is used for our own video streaming algorithms.

The main types of data that are communicated between components in a networked video streaming application are video-data, metadata and control-data. The framework supports these data types. The metadata and control data are assumed to be event-based and are communicated using the XMPP protocol itself. This protocol is XML-based and can therefore be used to send any data type that can be represented by structured text. Streaming video is done outside the protocol. The ability to communicate these data types is sufficient for the application example in Fig. 3. In general, these data types are sufficient for almost any network video streaming application.

For video streaming three types of communication are used dependent on the relative location of the components that are connected to each other:

- *Local*: For docks that are instantiated on the same host, shared memory is used for communication. The dock manager manages this shared memory.
- *RTP*: For connections between docks that reside within the same LAN the RTP protocol is used. Using RTP upon UDP makes it makes possible to meet real-time requirements because the protocol will not wait for lost packages.
- *SOCKS 5*: For connection between docks that reside on distinct LAN's (and therefore needs to traverse a NAT-router or firewall) no RTP connection can be setup because this protocol is IP-address based and hosts behind a NAT-router do not have an unique IP-address. Furthermore, firewalls could block the ports used by the protocol. A SOCKS 5 [9] proxy is used to setup a SOCKS 5 byte-stream between the two components. Such a byte-stream is based on a TCP connection and is therefore not very suitable for applications subject to QoS.

Typically, the real-time part of video content analysis applications resides on a LAN, whereas WAN connections are, due to their higher delays, mostly used for monitoring, control and notifications. For the latter tasks guaranteed delivery is more important, which makes a SOCKS 5 byte-stream a suitable candidate for inter-LAN connections.

For metadata communication, the XMPP protocol is used, except for intra-host communication, for which we use method invocation. The message passing XMPP protocol needs an XMPP server to relay messages between hosts. End-to-End connections can be used for intra-LAN communication of metadata but this requires an extension of the XMPP protocol (XEP-0246). Because the vast majority of the data communicated within a typical video content analysis application is video data, there is little to gain and therefore, this extension is not implemented.

Fig. 4 depicts a possible network structure supported by the framework. End-to-End RTP sessions are used for intra-

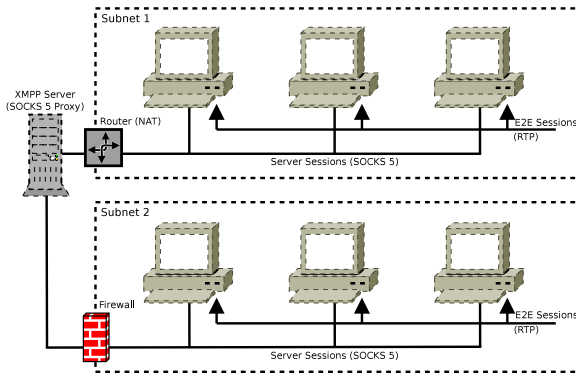


Fig. 4: Possible network structure supported by the ViFramework. End-to-End RTP sessions are used for intra-LAN video streaming. For inter-LAN video streaming the XMPP server is used as SOCKS 5 proxy. Only the XMPP server requires a public IP-address. Metadata communication is not depicted in this figure.

LAN video streaming whereas the XMPP server can be used as a SOCKS 5 proxy in order to setup a SOCKS 5 byte-stream between two hosts in different subnets. The XMPP Server needs to be accessible from both subnets so a public IP-address is required.

4.2 Interface definition

The typical pipe and filter architecture pattern [10] found in video streaming applications consists of an in- and output component with one or more intermediate processing components. The need for dynamic reconfiguration calls for a data-centric composition technique. The proposed framework allows dock builders to specify what data types the dock requires and provides. When deploying an application the required docks are instantiated. An interface-matching algorithm is used to calculate, given an interface, which interfaces can be connected to it. The result of this algorithm can be used to automatically set up an application or can aid the user in manually setting up the application.

The demand for flexible dock definitions and the use of the XMPP protocol makes XML an appropriate language for dock and interface definitions and it is therefore used as the *Interface Definition Language* (IDL) in the proposed framework. At design-time, a configuration XML file is designed for each host. At start-up, this file is read by the dock manager which parses, amongst others, its identifier, the XMPP server address and the available dock definitions from this file. A dock definition contains a dock identifier, a functionality description and a list of interfaces with their respective QoS properties. For each dock instantiated by the dock manager, a copy of the dock definition is made, which can be modified by the dock manager. Changes to these description instances can be made to, for example, bind interfaces by adding target information to the interface element or to set QoS properties.

Each video streaming interface can set a topic, which

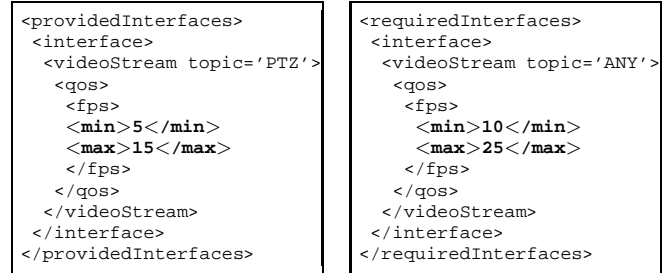


Fig. 5: Interface definition of two matching video streaming interfaces. The provided interface can operate in the range specified by the required interface. The required interface will accept streams with any topic.

can be used for stream identification and a number of QoS parameters. Two examples of interface definitions are listed in Fig.5. The matching algorithm checks whether the provided interface can meet all demands of the required interface which is the case in this figure. Metadata interfaces are defined by the XML representation of the object they communicate. The dock builder is able to construct any data type for communication as long as it is representable in XML. When, for example, the dock builder needs information about detected cars to be communicated, interface definitions as depicted in Fig. 6 can be specified. When a dock with a provided interface having this specification sends data, it fills the `<carInfo>` element with data and sends it to the required interface. This representation allows easy creation of new object types and easy extension of existing ones.

At run-time a dock can be easily replaced by an other dock with compatible interfaces but with potentially different functionality. The user receives an overview from the framework on what connections can be made between instantiated docks. With the proposed framework, creating more complex component graphs is quite straightforward as provided interfaces are able to setup connections to multiple required interfaces. When streaming video, each of these connections can have its own QoS properties. Moreover, this allows run-time extension of existing applications by adding additional processing steps, or by branching the video stream at a certain point, in order to create a separate processing path.

4.3 Host failure recovery

The use of the XMPP protocol as a network substrate provides the proposed framework with information about the presence of dock managers. The XMPP server will notify the distribution manager when a host has gone off-line. The distribution manager will react on such an event by starting a recovery algorithm. This algorithm tries to re-instantiate the docks that were running on the failing host, on other (possibly unused) hosts, tries to reconnect them and upon success, restarts the failed part of the application. An

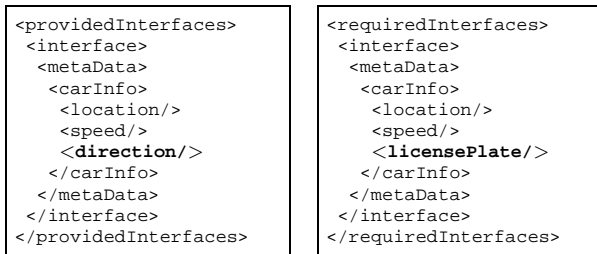


Fig. 6: Interface definition of two non-matching metadata interfaces. The addition of the `<direction/>` element is allowed because a provided interface might supply more data than needed. Adding the `<licensePlate/>` element in the required interface will make this interface no longer matching because the provided interface can not provide this element.

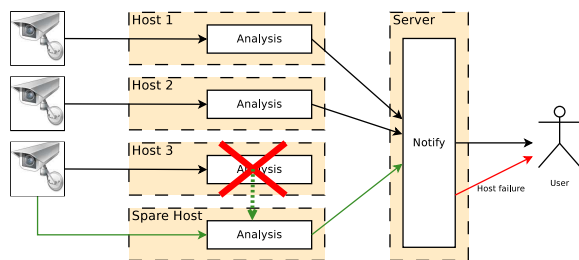


Fig. 7: Crash recovery example. Three camera feeds are processed, each by a dedicated host. All data gathered by the analysis algorithms are forwarded to the server which will notify the end-user on certain events. If *Host 3* fails, the framework will try to find a host to re-instantiate the lost analysis dock on and reroute the video stream that was processed by the crashed host through the new one. The end-user will be notified about this host-failure.

example situation is depicted in Fig. 7.

The framework is designed for real-time systems and therefore no attempt will be made to resend frames that are lost due to host-failure. Because the framework is targeted at surveillance applications, crash recovery should be performed in the least amount of time possible in order to accomplish minimal data-loss.

4.4 Resource Management

To enable automatic deployment of new applications, dynamic reconfiguration of existing application and host failure recovery the distribution manager needs information about the available resources (e.g. CPU, memory, network bandwidth) on each connected host. To enable load balancing, also information about the current resource usage is required from each connected host. Resource information is gathered by the dock managers and forwarded to the distribution manager. This enables resource management on two levels; at host-level and at system-level.

Ideally, the dock manager process is the only process running on each host apart from mandatory OS processes. Because of the low resource usage of OS processes it can be assumed the dock manager has all the host's resources at its disposal. As future work, this could be forced by running

the dock manager in a virtual machine. The dock manager will spawn a new thread for each dock it instantiates. At this point resource reservations can be made for this new dock. Docks are allowed to spawn new threads themselves. In the current implementation, host-level resource management is left to the operating system.

At system-level, the distribution manager has knowledge about the available and used resources of each host. Therefore, it can make educated decisions when deploying new docks. For example, when the new analysis dock is instantiated in the host failure recovery situation of Fig. 7, the distribution manager will opt for the unused host, rather than a host that is already doing heavy computation.

Resource requirements for video content analysis algorithms are often data-dependent [11]. This requires the framework to respond to a sudden increase in resource requirements. If a host cannot meet the resource requirements of its docks, the distribution manager needs to redistribute the application in a more appropriate way.

So, to make its global deployment decisions, we see that the distribution manager needs resource information that is as accurate and recent as possible. As stated in [12], large applications that constantly send resource information to a central service create an extensive network usage overhead. This makes an implementation in which the dock managers send resource usage updates to the distribution manager at a high fixed rate not well scalable and therefore unsuitable for a video-streaming framework. In [12] a solution for this problem is proposed. This solution divides resource usage in three usage-levels and only sends on level transitions. The ViFramework uses a similar, but more extensive solution to solve this problem.

For each resource (e.g. CPU usage) a new value will only be reported if it exceeds a user-defined threshold with respect to the last reported value and only when this situation persists for a user-definable duration. Fig. 8 shows an example resource graph. This solution enables a trade-off to be made by the end-user of the framework between network bandwidth usage and information granularity. It is an improvement over [12] because it provides the end-user with more detail when needed and it prevents large data bursts when resource usage oscillates between two usage-levels.

5. Framework Evaluation

In order to evaluate the proposed framework, the computational overhead and the time needed for host failure recovery were measured. Because data compression and streaming, although configurable by the framework are not dependent on the framework, no network usage measurements for applications deployed on multiple hosts are carried out. Furthermore, after application initialization, the only framework related network traffic is resource usage information, which is negligible.

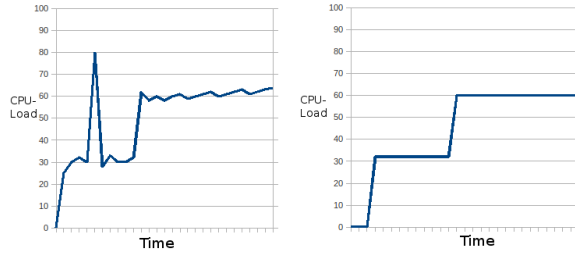


Fig. 8: Left: Host CPU usage registered by the dock manager | Right: Resource graph received from the dock manager at the distribution manager.

Table 1: Overhead Measurements

Cpu usage	Min	Avg	Max
Standalone1	58%	60%	62%
Frameworked1	65%	67%	70%
Standalone2	84%	87%	89%
Frameworked2	95%	97%	99%

5.1 Evaluation method

An application was created that reads a video from file, applies *object detection* on the video and writes the resulting video to a display. The video used has a resolution of 640x480 at 15 frames per second. For framework evaluation the application is divided into three docks (read, process, write) which are all deployed on the same host. The frameworked application and the standalone application were both executed on the same host.

The test host contains a quadcore Intel® Core™ i7 870 processor at 2.93 Ghz with 2 GB of RAM. The operating system used is Linux 2.6.36-26.

5.2 Overhead

Video content analysis algorithms are most often computationally intensive, making it important that the framework overhead in terms of CPU usage is minimized. Furthermore, because of real-time requirements, the processing delay the framework introduces should also be minimal. To measure the overhead the framework imposes, a standalone application is compared to the same application in the proposed framework but deployed on only one host. While the algorithm was running, the CPU usage was measured for three minutes. Two runs were made, the second run executing a more demanding version of the object detection algorithm.

The results of the measurements are presented in Table 1. Both runs indicate a framework overhead of about 12%. For most target applications this overhead is considered acceptable, and can be improved as the current framework implementation is still a rapid prototype.

Table 2: Failover Measurements

Metric:	Value:
# Measurements	20
Min. time	467 ms
Avg. time	584 ms
Max. time	830 ms
Avg. frame-loss	9

5.3 Failover automation

On host failure the framework tries to re-instantiate a failing dock as quickly as possible in order to lose a minimal amount of data. Because of real-time requirements, no data is retransmitted, so the longer it takes to take over the functionality of the failing host, the more data will be lost. In this benchmark the three docks of the evaluation application were deployed on two hosts. The reading and writing docks are deployed on one host and the object detection dock is deployed on the other. The dock performing object detection is deliberately interrupted by killing its dock manager process. The time between this point and the point where the second host has taken over the dock on the failed host is measured in order to calculate data-loss. The results are presented in Table 2 and show that fast recovery is possible using the proposed framework. Losing slightly more than half a second of data on average is acceptable for most surveillance applications.

6. Related Work

In [5] the advanced component system SOFA 2.0 is presented which was created in order to overcome limitations of formerly existing component-based systems. Due to the lack of video streaming support and the service-oriented nature of SOFA 2.0 this component system is considered unsuitable for real-time video streaming applications. Nevertheless, this work inspired some aspects of the proposed framework such as docks being instrumented components and the dynamic re-configuration of deployed applications.

In [13] the OpenDDS component framework is presented which supports complex data flows and dynamic reconfiguration. The drawbacks of OpenDDS are; the lack of video streaming support, the absence of security algorithms and problems with NAT router and firewall traversal. Another problem is the inflexibility of the framework when designing components for rapid prototyping, dynamic data types, for instance, are not supported.

In [14] the GStreamer framework is presented that focuses on audio and video streaming applications. The framework aims at creating single machine multimedia applications by composing existing components called plug-ins. The framework lacks presence information which is a main feature of the proposed framework and has no built-in means that support dynamic reconfiguration.

In [8] a network substrate for desktop grid computing named *Orbweb* is presented. This work describes the effort that is made to extend the XMPP protocol in order to meet the substrate needs. This substrate uses XMPP for NAT and firewall traversal and takes advantage of the available security protocols embedded in XMPP. *Orbweb* is considered unsuitable because no functionality for real-time applications is available.

In [6], Westerink proposes a flexible framework for building multi-media streaming applications. This framework identifies the general architectural structure of streaming applications and using this knowledge to create an easy-to-use framework which is used for some existing applications. The framework proposed by Westerink is targeted at creating single machine applications from existing components, and therefore not suitable to be used as a networked component framework.

7. Conclusion

In this paper the ViFramework, a framework for networked video streaming components targeted at surveillance applications, is presented. This framework provides dock component builders with flexible means to specify docks using XML as definition language. The end-users are provided with easy-to-use tools to create complex application architectures from the available docks. The combination with the XMPP protocol enables the framework to deploy an application on a WAN by using firewall and NAT-router traversal. This enables remote monitoring, control and notifications. Basic resource monitoring on host-level is performed. Gathered information is communicated to a distribution manager in a smart and configurable manner in order to enable application wide load balancing.

A failover algorithm enhances the robustness of the frameworked application. Evaluation of this algorithm shows that failover is achieved within acceptable time. Measurements show that framework has an acceptable computation overhead. Overall the measurements of the presented prototype implementation show that it is efficient and suitable for video surveillance applications.

8. Future work

Future functionality of the proposed framework will include resource usage profiling of docks on the available hosts as is done by Korostelev et. al in [15]. This will enable the distribution manager to predict what resources a certain dock will use on a host. Using this information the distribution manager can deploy applications more efficiently. The dock manager will also perform resource allocation instead of the operating system which is responsible for this in the current implementation.

The current implementation only supports end-to-end connections. More interface types are to be developed to enable

other communication constructs such as publish-subscribe and multi-cast.

For now it is assumed that all docks are pre-compiled on the hosts used by the framework. A *Dock Repository* will be developed that allows run-time uploading of docks to hosts. This facilitates adding “blank” hosts to the system on which, on demand, appropriate docks can be installed.

Optimizations to the framework can be made in order to reduce CPU usage overhead.

Acknowledgment

The research reported in this paper has been done in the context of the first author's master's project. The project has been carried out at ViNotion B.V. and the support received from the company and its staff is gratefully acknowledged. Furthermore, we thank Johan Lukkien and Egbert Jaspers for their comments on an earlier version of this paper.

References

- [1] G. T. Heineman and W. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Professional, June 2001.
- [2] CANTATA, “Content aware networked systems towards advanced and tailored assistance,” URL, 2011, <http://www.hitech-projects.com/euprojects/cantata/>.
- [3] ViCoMo, “Visual context modeling,” URL, 2011, <http://www.vicommo.org/>.
- [4] B. Kersten, “Instrumentation of networked video streaming components (to appear),” Master's thesis, Eindhoven University of Technology, April 2011.
- [5] T. Bures, P. Hnetyinka, and F. Plasil, “Sofa 2.0: Balancing advanced features in a hierarchical component model,” *Software Engineering Research, Management and Applications, ACIS International Conference on*, vol. 0, pp. 40–48, 2006.
- [6] P. Westerink and F. Schaffa, “A high level flexible framework for building multi-platform multi-media streaming applications,” in *Wireless and Optical Communications Conference (WOCC), 2010 19th Annual*, May 2010, pp. 1–5.
- [7] XMPP, URL, 2011, <http://www.xmpp.org/>.
- [8] S. Schulz, W. Blochinger, and M. Poths, “Orbweb - a network substrate for peer-to-peer desktop grid computing based on open standards,” *J. Grid Comput.*, vol. 8, no. 1, pp. 77–107, 2010.
- [9] “Socks protocol version 5,” URL, 2011, <http://tools.ietf.org/html/rfc1928>.
- [10] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Apr. 1996.
- [11] I. David, B. Orlic, R. H. Mak, and J. J. Lukkien, “Towards resource-aware runtime reconfigurable component-based systems,” *Services, IEEE Congress on*, vol. 0, pp. 465–466, 2010.
- [12] L. Rizvanovic and G. Fohler, “The matrix - a framework for real-time resource management for video streaming in networks of heterogeneous devices,” in *The International Conference on Consumer Electronics 2007*, January 2007. [Online]. Available: <http://www.mrtc.mdh.se/index.php?choice=publications&id=1164>
- [13] OpenDDS, URL, 2010, <http://www.opendds.org/>.
- [14] GStreamer, URL, 2011, <http://gstreamer.freedesktop.org/>.
- [15] A. Korostelev, J. Lukkien, J. Nesvadba, and Y. Qian, “Qos management in distributed service oriented systems,” in *Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference: parallel and distributed computing and networks*, ser. PDCN'07. Anaheim, CA, USA: ACTA Press, 2007, pp. 345–352. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1295581.1295637>

Computing the Configuration Space Using Arrays with Reconfigurable Optical Buses

John Jenq

Department of Computer Science, Montclair State University, Montclair, NJ 07043, USA

Abstract - Configuration space computation is a transformation which can be adopted in robot path planning. This process reduces a robot with dimensions to become a single reference point by expanding each obstacle on the image plane. This approach reduces a complex problem into a simple one. In this paper parallel algorithms for computing the configuration space obstacles by using arrays with reconfigurable optical buses (AROB) is presented. The digitized images of the obstacles and the robot are stored in an image plane. These algorithms take $O(1)$ cycle time and are optimal.

Keywords: Configuration space, robotics, parallel processing, reconfigurable networks, optical computing

1 Introduction

Arrays with reconfigurable systems had been intensively studied in the past. By using optical bus (or waveguide) as communication means, parallel algorithms were developed to solve various fundamental operations. Operations such as matrix manipulation problems, sorting, selection, data routing were developed [4] [9] [13] [14] [15] [17] among many others. There are different models and variations for reconfigurable network optical computers see for example [3][15][18]. The array with reconfigurable pipelined bus system (LARPBS) and arrays with reconfigurable optical buses (AROB) had been intensively studied. In this report, we developed algorithms for 2D arrays with reconfigurable buses (2D AROB). Constant time algorithms for computing the configuration space on array of reconfigurable optical buses (AROB) was developed.

Computing the configuration space is an important problem in path planning for robotics applications. The objective of path planning is to find a path to move a robot A from a position s (the initial position) to another position d (the final position) without colliding with the obstacles already in space R . One way to solve this problem is the configuration space obstacle approach (for example [8][10][11]) which reduces the robot A to a single reference point p and expands each obstacle B_j to include all the positions of p that cause a collision between A and B_j . The expansion of an obstacle B_j is called the configuration space obstacle of B_j . In the new representation, the object A (robot) becomes a single point. The configuration space therefore reduces a complex problem into a simple one.

Figure 1 shows a robot A and two obstacles (B_1 and B_2). To compute the configuration obstacle of an obstacle of B_j , first invert the robot A , i.e. to rotate A about the reference point by 180° and then slide the reference point around the boundary of the obstacle B_j . The union of the area covered by A during the sliding, and the area originally covered by B_j defines the configuration space obstacle of B_j . Note here the orientation of A does not change when we move A around the obstacles. In Figure 1 the area enclosed by the dark lines are the configuration space obstacles of B_1 and B_2 . S designates the source and D designates the destination positions.

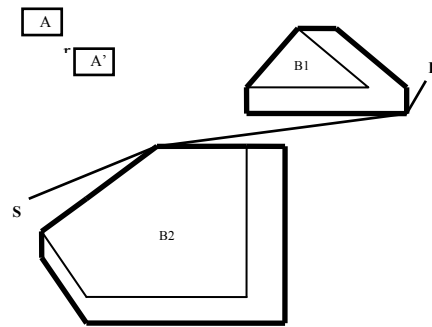


Figure 1. Robot A and its inverted A' , dark lines form the configuration obstacles of B_1 and B_2 .

Parallel processing approaches to solve path planning had been developed. Tzionas, Thanailakis, and Tsalides presented parallel algorithm for collision free path planning of a diamond-shaped robot and its implementation in VLSI[19]. Jenq, et. al. developed optimal RMESH algorithms[7], their algorithm requires $N \times N \times D$ RMESH, Jenq and Li also developed optimal algorithms of computing the configuration space using hypercube computers[5][6]. Their algorithms run in $O(\log N)$ time for an $N \times N$ image by using $N \times N$ processors and are optimal for hypercube computers. Dehne, Hassenklover, and Sack presented a systolic algorithm for computing the configuration space obstacles in a plane for a rectilinear convex robot [1]. Their algorithm takes $O(N)$ time for an $N \times N$ image on an $N \times N$ mesh computer.

The digitized bitmap images of convex robots are rectilinear convex polygons. The converse statement may not be true. A polygon is rectilinear convex if (1) the polygon is formed by horizontal and vertical line segments, and (2) the intersection of the polygon with any horizontal or vertical line consists of

at most one line segment. Figure 2 depicts a polygon which is rectilinear convex.

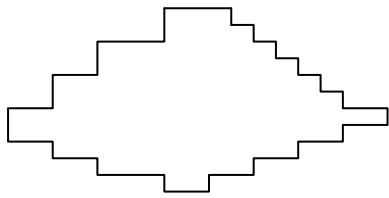


Figure 2 A rectilinear polygon

In this paper, algorithms for computing the configuration space obstacles for robots with shape of convex polygons were presented. The algorithm can be modified and easily applies to robot of shapes circular and rectangular. We omit the details of these algorithms here. Basic data movement operations related to AROB was developed. These algorithms take constant time for an image on an AROB with $N \times V$ processors.

We organized this paper as the following. In section 2, the basic configuration of AROB is discussed. In section 3, we present fundamental data movement operations which are used to develop the optimal algorithm. Section 4, the configuration space algorithm on AROB is presented. Section 5 concludes the report.

2 Preliminaries of AROB

One dimensional AROB is similar to one dimensional array with pipelined buses 1D-APPB[2]. The 1D AROB allows each processor to set its own switches (receiving and transmitting) to connect to the bus. A switch can be set to either cross or straight. For example, to partition a 1D AROB into two independent 1D AROBs, processor P_i can set both of its switches to cross. One 1D AROB contains P_0 to P_{i-1} and the other from P_i to P_{N-1} ; where N is the size of the original 1D AROB. For 2D AROB, each processor has index of (i, j) . There are four switches for each processor. Figure 3 shows a 4×4 2D AROB. Figure 4 shows the permissible switch setting for 2D AROB.

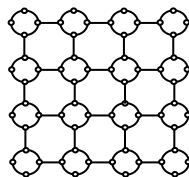


Figure 3 A 4×4 AROB

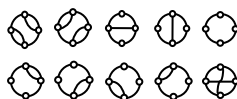


Figure 4 Permissible switch setting for 2D AROB

3. Fundamental Data Manipulation Operations

In this section we define several basic data manipulation algorithms for AROB. These operations are served as the building blocks to construct algorithm for configuration space obstacle computation in the next section.

Broadcast

This is the most basic operation in a reconfigurable mesh. In a data broadcast operation, data originated in one PE are sent to all the PEs connected to the bus. All PEs who want to fetch the transferred data can receive and read the content of the optical bus when the selection signal arrives. The broadcast operation takes $O(I)$ time [12].

Translate

Each $PE(i, j)$ has data in it's $A(i, j)$ variable that is to be shifted circularly to $B(i, j)$ variable of a processor that is (x, y) away, where x and y are a pair of vertical and horizontal displacements. The translate operation will move A circularly to up and left by x , and y positions. I.e., $A(i, j)$ will be translated to $B((i - x + N) \% N, (j - y + N) \% N)$. This operation is called *Shift* in [20].

Shift

Each PE has a data in it's A variable that is to be shifted to B variable of a processor that is s units, $s > 0$, to right or left in the same row (column). A variant of shift is the operation of *circular shift*, which performs shift with wrap-around. These operations can be done in $O(I)$ time. Note it is a special case of Translate.

Routing

Consider 1D AROB routing problem. When $O(I)$ packets originate from any node and $O(I)$ packets are destined for any node. This problem can be resolved in $O(I)$ cycles[16]. In [16] it is call h -relation operation. When h is 1 it is partial routing (I -relation).

Inversion

This operation rotates a rectilinear polygon by 180° about a reference point (i, j) , where i and j are integers in the range of $0 \dots N-1$. Assuming that the gray values of the image of the rectilinear polygon are the same (e.g., all are of value 1), this operation can be done in constant time on an $N \times N$ AROB. The algorithm is listed in Figure 5.

-
- Step1** Reference point broadcasts its i and j indices to all pixels of the robot.
 - Step2** Each robot PE applies two shift operations: *Shift* left and *Shift* right with $A=1$;
 - Step3** PE that did not receive two 1s are boundary pixels; boundary pixels compute $(newI, newJ)$ index after the inversion. Boundary pixel PE identifies itself as left or right boundary PE.

- Step4** Right boundary *PE* set its $A=1$, left boundary *PE* set its $A=2$
- Step5** Perform Routing : boundary pixels send information of its A to *PE* of index $(newI, newJ)$.
- Step6** *PE* receives 1 is a left boundary pixel of the inversion robot; *PE* receives 2 is a right boundary pixel of the inversion robot
- Step7** Setup row buses; new left *PE* broadcasts $A=1$; *PE* that receive 1 is part of the inversion robot

Figure 5. A constant time inversion operation of a rectilinear polygon.

Step3 is a simple logic for robot *PE* to test if it's a boundary *PE*, here we assume there is no robot with diameter of N . otherwise either extra artificial *PEs* need to be introduced to enclose the image plan, or small modification of the algorithm is required. It is easy since if a *PE* with j index of 0 or $N - 1$ would be in the boundary. Step5 performs the routing operation its complexity is $O(1)$.

PrefixSum

In [16], Pavel and Akl presented a positive integer prefix sum operation on 1D AROB. The prefix sums of n integers v_i with $0 \leq v_i \leq n, 0 \leq i \leq n - 1$, and $\sum_{i=0}^{n-1} v_i \leq n$, can be computed in $O(1)$ steps. In this report we extend their algorithm to cover the condition when there are negative integers as well. For $0 \leq |v_i| \leq n, 0 \leq i \leq n - 1$, and $\sum_{i=0}^{n-1} v_i \leq n$ if $v_i \geq 0$ and $\sum_{i=0}^{n-1} (-v_i) \leq n$ if $v_i \leq 0$, the prefix sums of n integers v_i can be done in constant steps.

- Step1** For *PE*(i) with $v_i \geq 0$, for $0 \leq i < N$ perform positive integer prefix sum operation of values v_i
- Step2** For *PE*(i) with $v_i \leq 0$, for $0 \leq i < N$ perform positive integer prefix sum operation of values $-v_i$
- Step3** For *PE*(i) with $v_i \geq 0$, for $0 \leq i < N$ perform Even-Odd-Ranking
- Step4** Even rank *PE* and the odd rank *PE* immediate follows it form independent bus with even rank *PE* as leader and odd rank *PE* as END, do broadcast of its $A(i)$
- Step5** All *PEs* read the value broadcast and set it to $B(i)$ except for END *PE* which $B(i)$ set to its $A(i)$
- Step6** Odd rank *PE* and the even rank *PE* immediate follows it form independent bus with odd rank *PE* as leader do broadcast of its $A(i)$
- Step7** All *PEs* read the value broadcast and set it to $B(i)$ except for END *PE* which $B(i)$ set to its $A(i)$
- Step8** Repeat Step3 to Step7, replace $B(i)$ with $C(i)$ for *PE*(i) with $v_i \leq 0$, for $0 \leq i < N$
- Step9** Prefix-Sum $(i) = B(i) + C(i)$

Figure 6. A constant time Prefix Sum operation of 1D AROB

Odd-Even-Ranking

Each *PE*(i) has a flag $selected(i)$, which is set to true if *PE*(i) is selected. A Ranking operation assigns a rank to each *PE*,

where the rank of *PE*(i), $rank(i)$, is the number of selected *PEs* whose indices are less than i . Note it takes $O(\log N)$ time on RMESH. The Even-Odd-Ranking is a special case of Ranking. If the $rank(i)$ is an odd number then assign 1 to it, otherwise assign 0 if $rank(i)$ is even. If we assign 1 to *PE* with $rank(i)$ even, and 0 to $rank(i)$ odd then we call it Odd-Even-Ranking. Note also that Ranking is a special case of positive integer prefix sum operation in [16]. By assigning the values (v_i) to be summed with integer value of 1, it can be easily done. Note the result prefix sum value minus 1 is the rank of the *PE* assuming rank start with 0 rather than 1. This operation takes $O(I)$ time on AROB.

4 Computing configuration space on AROB

In this section, an $O(1)$ time algorithm that computing configuration space obstacles on AROB for a WBP is presented. A WBP (well behavior polygon) robot is defined as in [6]. Briefly speaking, a WBP is a polygon that can be partitioned into at most four L-shaped polygons as shown in Figure 12. See [6] for the detail discussion about this type of polygons. The intersection of the two dotted lines is called the base point. It is possible that for some cases there are might be two base points on the WBP as an example shown in Figure 7(b), By applying the translate operation on obstacles, one can make it works as if we only have one base point. See detailed description in [6].

To compute the configuration space obstacles for a robot with the shape such as the one shown in Figure 7(a) can now be reduced to the case as if the robot is of L-shaped. Simply apply four iterations of the configuration space obstacle computing algorithm for L-shaped robots, and union the areas, one can compute the final configuration space obstacles. Since these four procedures are similar, we present only one type of the L-shaped polygon here. Without loss of generality, let's consider an L-shaped polygon as show in Figure 7, where r is the base point.

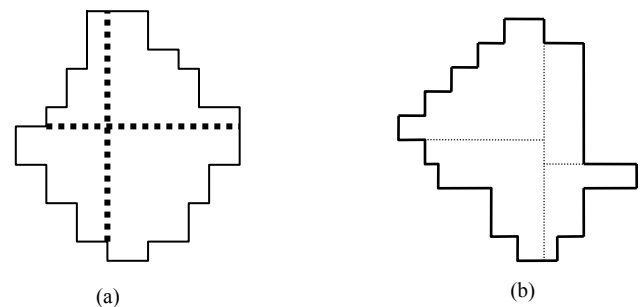


Figure 7 (a) A WBP convex robot converts and partition into four L-shaped rectilinear polygons
(b) AWBP with two base points

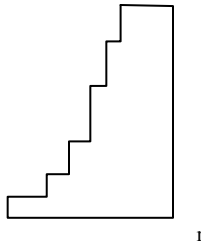


Figure 8 An L-Shaped polygon

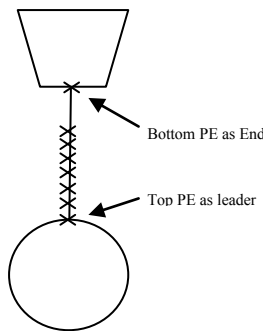


Figure 9 Vertical grow of obstacles

We partition the growing of the obstacles into two phases. The vertical expansion phase and horizontal expansion phase. In the vertical expansion phase, we draw a vertical line from any boundary pixel of obstacles. This vertical line then will grow horizontally to its left for the L-shaped polygon in Figure 8. Note the horizontal length exhibit dominant property of [5]. Consider scenery as in Figure 8. Let's define a top PE as the PE which is on "top" of an obstacle. A bottom PE can be defined similarly as a boundary PE with other obstacle PE on top of it. Note a boundary PE under a top PE did not grow a vertical line instead it only grow one horizontal line segment during the horizontal growing phase due to dominant property. Figure 10 outline an algorithm to grow obstacles vertically.

-
- Step1** Top PEs set as leader PE
 - Step2** Bottom PEs set as End PE
 - Step3** Top PEs broadcast its row index t
 - Step4** PEs (i,j) on the column bus receive this value and compute $SegNumber = t-i$;
 - Step5** if $SegNumber \leq h$ then $mark(I,j) = true$; else $mark(I,j) = false$;
-

Figure 10 A constant time Fill-up operation on AROB

Note after Step5, the obstacles had been expanded vertically. Before grow the obstacles horizontally, all newly grown PEs need to know its runLength by using its

SegNumber as an index to look up the corresponding runLength information of the L-Shaped polygon. For a diameter $O(1)$ robot, this information can be broadcast from robot boundary PEs. Assuming there are h segments. In $O(h)$ time, all expanded obstacles along with the original obstacle boundary PEs can receive its runLength information.

Each expanded obstacles PEs along with all the boundary PEs will then perform the horizontal growing phase as shown in Figure 11.

-
- Step1** Fill-up
 - Step2** Broadcast segment runLength information
 - Step3** For boundary obstacle PEs and expanded obstacles PEs calculate destination PE's j index $d_j = j - RunLength$ and set its $A(I,j)$ to be 1
 - Step4** Perform Routing (h -relation) operations
 - Step5** For PE receives runLength information add -1 to its A variable.
 - Step6** Perform Integer Prefix Sum operation
 - Step7** PE with $A(I,j) = 1$ is the leader and $A(I,j) = 0$ as the End setup row bus
 - Step8** Leader PEs broadcast its $-A(-1)$ value and all PEs in the subbus receive and mark it as configuration space obstacle PE
-

Figure 11 Computing of configuration space for L-shaped robot on AROB: horizontal phase

Step1 of Figure 11 starts the vertical expansion. Step2 allows all expanded obstacle PEs to receive the runLength information. Step3 calculate the destination PE index. Step 4 perform h -relation operation to route -1 value to the other end of the segment. Use of -1 is to preparing for the prefix sum operation. Step 6 perform prefix sum operation. Between 1 and 0 is a segment that shall mark as newly created configuration space of obstacles. Figure 12 shows an example of the Routing results.

(a)										
(b)	-1	-1		-1	1	1	1	-1		1
(c)	0	-1		-2	3	2	1	0		1
(d)	1	1	1	1	1	1	1	1	1	1

Figure 12 Horizontal Expansion of configuration space obstacles

Figure 12 (a) shows there are four segments to be joined together. Here we use $[$ to indicate the left end point of a segment. The $]$ indicates the right end point of a segment. The other symbols such as star were used to pair corresponding end points together to form a segment. Let's assign 1 for starting (right) point of a segment and -1 the end (left) of the segment as shown in (b). In (c) the prefix sum operation is performed. PE with value 1 is the start of the final combined segment and 0 is the end of the final combined segment. We are suppose to

draw any combined segment by filling 1s between start end point and its corresponding end point. The start segment PE is the leader and PE with 0 is the End of independent bus. A broadcast operation transmits the value 1 to all the PEs in the bus. It fills all the PEs between leader and end PEs with 1 which is the horizontal configuration space obstacles. Because the complexity of all steps in Figure 11 is $O(1)$ therefore the total complexity of computing the configuration space for L-shaped robot on 2-D AROB is $O(1)$. It implies that the total complexity to compute the configuration space for a WBP shape robot on 2-D AROB is also $O(1)$.

5 Conclusion remarks

Basic data movement operations are discussed and developed for robot path planning on AROB. Algorithm for computing the configuration space obstacles for WBP convex robots by using 2D-AROB was developed. This algorithm runs in constant time and uses constant space and therefore is asymptotically optimal.

6 References

- [1] F. Dehne, A. Hassenklover, and J. Sack, "Computing the configuration space for a robot on a mesh-of-processors," Proceedings 1989 ICPP. vol. 3, pp. 40-47, 1989.
- [2] Z. Guo, R. Melhem, R. Hall, D. Chiarulli, and S. Levitan, "Pipelined communications in optically interconnected arrays," *Journal Parallel Distributed Computing*, vol. 12, no. 3, pp 269-282, 1991.
- [3] Z. Guo, Optically Interconnected Processor Arrays with Switching Capability. *Journal of Parallel and Distributed Computing*. 23, 1994, pp 314-329.
- [4] M. He, X. Wu, S. Zheng, and B. Englert, "Optimal Sorting Algorithms for a Simplified 2D Array with Reconfigurable Pipelined Bus System", *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 3, pp303-312, 2010
- [5] J. Jenq and W. Li, Optimal Hypercube Algorithms for Robot Configuration Space Computation, Proceedings of the 1995 ACM Symposium on Applied Computing, pp 182-186
- [6] J. Jenq and W. Li, Computing the Configuration Space for a Convex Robot on Hypercube Multiprocessors, Proceedings of the 7th IEEE Symposium of Parallel and Distributed Processing, pp 160-167, 1995
- [7] J. Jenq, D. Wang, and W. Li, Computing the Configuration Space on Reconfigurable Mesh Multiprocessors, Proceedings of International Conference on Parallel and Distributed Computing Systems, 2000, pp 186-191
- [8] L. Kavraki, "Computation of Configuration-Space Obstacles Using the Fast Fourier Transform", *IEEE Transactions on Robotics and Automation*, vol. 11(3), pp 408-413, 1995
- [9] K. Li, Y. Pan, S. Zheng, "Fast and Processor Efficient Parallel Matrix Multiplication Algorithms on a Linear Array With a Reconfigurable Pipelined Bus System", *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 8, 1998, pp. 705-720.
- [10] T. Lozano-Perez and M. A. Wesley, "An algorithm for planning collision-free paths among polyhedral obstacles," *CACM*, pp. 560-570, 1979.
- [11] T. Lozano-Perez, "Spatial planning: A configuration space approach," *IEEE Trans. on Computers*, pp. 108-120, 1983.
- [12] Y. Pan, "Order Statistics on Optically Interconnected Multiprocessor Systems", First International Workshop on Massively Parallel Processing using Optical Interconnections, pp162-169, 1994
- [13] S. Pavel and S.G. Akl, "Matrix Operations Using Arrays with Reconfigurable Optical Buses", *Journal of Parallel Algorithms and Applications*, 8, 1996, pp 223-242
- [14] S. Pavel and S.G. Akl, "Sorting and Routing in Arrays with Reconfigurable Optical Buses", Proceedings of International Conference on Parallel Processing, 1996, vol. 2, pp 90-94.
- [15] S. Pavel and S.G. Akl, "On the Power of Arrays with Reconfigurable Optical Bus," Proc. Int'l Conf. Parallel and Distributed Processing Techniques and Applications, 1996, pp. 1443-1454.
- [16] S. Pavel and S.G. Akl, "Integer Sorting and Routing in Arrays with Reconfigurable Optical Buses", *International Journal of Foundations of Computer Science*, vol. 9, no. 1, 1998, pp 99-120.
- [17] S. Rajasekaran, S. Sahni, "Sorting, Selection, and Routing on the Array with Reconfigurable Optical Buses", *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 11, 1997, pp. 1123-1132
- [18] S. Sahni, "Models and Algorithms for Optical and Optoelectronic Parallel Computers," *International Journal Foundations of Computer Science*, vol. 12, no. 3, pp. 249-264, 2001.
- [19] P. Tzionas, A. Thanailakis, and P. Tsalides, "Collision-Free Path Planning for a Diamond-Shaped Robot Using Two dimensional Cellular Automata", *IEEE Transactions on Robotics and Automation*, vol.13(2), pp 237-250, 1997
- [20] C. Wu, and S. Horng, "L2 vector median filter on arrays with reconfigurable optical buses", *IEEE transaction on Parallel and Distributed Systems*, vol. 12, no. 12, pp 1281-1292, 2001

Design and Optimization of Hybrid MD5-Blowfish Encryption on GPUs

Zhu Wang, Josh Graham, Noura Ajam, and Hai Jiang

Department of Computer Science, Arkansas State University, Jonesboro, AR 72467, USA

Abstract—*Nowadays, data has been playing an indispensable role in almost all industrial areas. Data integrity and security over Internet, other types of media and applications have become the major concerns in computer world. If confidential or sensitive data is forged, juggled or wiretapped by an attacker, capital losses might occur. Encryption is one of the major mechanisms to prevent this from happening. So far, there are a variety of encryption algorithms, but none of them excels at both efficiency and high security. This paper proposes a hybrid, efficient and parallel cryptographic algorithm, MD5-Blowfish, on GPUs. This new scheme can bring in strong cryptographic effects without much performance degradation. Experimental results have shown how different GPU configurations and optimizations influence the overall performance. Based on these results, the best system configuration can be selected. In experiments, the new algorithm has demonstrated its effectiveness and efficiency in data integrity and encryption process.*

Keywords: Blowfish, MD5, GPU, CUDA

1. Introduction

Information security, computer security, and network security all call for encryption to protect the message and provide data integrity as well as authenticity. With the sharp increase of information communication volume, security becomes a focal issue.

There are a tremendous number of different encryption algorithms and applications. MD5 (Message-Digest algorithm) is a cryptographic hash function, which is widely used in many applications. MD5 takes an arbitrary length of message as input and produces a 128-bit hash value as output. The typical application of MD5 is the data integrity checking. However, Bert den Boer and Antoon Bosselaers found “prepseudo-collisions” for the MD5 compression function in 1993[1]. It means two different initialization vectors which produce an identical digest, i.e., $MD5(M, N_i) = MD5(M', N_i)$. Without referring to theory, Xiaoyun Wang found many real collisions in 2004 and on IBM P690, it took about one hour to get such messages M and M' , while in the fastest cases it took only 15 minutes [2]. Vlastimil Klima also presented a new collision search algorithm to find collisions on a 1Ghz desktop PC in 4 hours [3]. These all indicate that MD5 is no longer secure.

Blowfish is a symmetric block cryptographic algorithm designed by Bruce Schneier in 1993 to replace the Data Encryption Standard (DES). Due to its good encryption rate in software, no effective cryptanalysis has been found to date [4]. However, it shares the common weakness of symmetric algorithms. If the secret key is discovered, all message can be decrypted. So the secret key needs to be changed on a regular basis. However, key replacement degrades Blowfish since the pre-processing of a new key is equivalent to encrypting about 4K bytes of text. Moreover, S.Vaudenay in 1996 [5] found out that weak keys of Blowfish can be detected and broken by the same attack with only certain size of known plaintexts. So it is insufficient to use a single kind of cryptographic algorithm in applications.

With rapid development of network technology and growth of information around the world, not only information security but also processing efficiency becomes important. Both symmetric and asymmetric key encryption schemes need to speed up without employing expensive dedicated cryptographic accelerators. In the meantime, the computing power of graphics processing unit (GPU) has significantly increased and has far surpassed the pace of CPU. Especially in the high performance computing area, applications using GPU gain manifold benefits due to its highly parallel structure.

This paper proposes a novel parallel cryptographic algorithm, merging and modifying from MD5 and Blowfish encryption schemes, which can enhance security. To reduce performance loss, this encryption algorithm is designed and developed based on NVIDIA CUDA (Compute Unified Device Architecture). This paper makes the following contributions:

- A hybrid MD5-Blowfish cryptographic algorithm is developed to overcome the weakness from symmetric block cryptographic and hash function schemes.
- A CUDA-based parallelization design is deployed to maintain high data encryption rate.
- Experimental results and performance analysis are provided to demonstrate the effectiveness and efficiency of the hybrid MD5-Blowfish algorithm.

The remainder of this paper is organized as follows: Section 2 gives an overview of related technology for MD5, Blowfish and GPUs. Section 3 describes the design and implementation of hybrid MD5-Blowfish algorithm. In Sec-

tion 4, performance analysis and experimental results are provided. Section 5 mentions some related work. Finally, our conclusion and future work are described in Section 6.

2. Background

2.1 MD5 (Message-Digest algorithm 5)

2.1.1 Algorithm Design

In MD5, the input message is broken up into chunks of 512-bit blocks (each with sixteen 32-bit sub-blocks). After a series of operations, MD5 produces a 128-bit message digest with four concatenated 32-bit blocks for the integrity of a file.

To compute the digest of a message, padding bits are appended first to make the message's length congruent to 448, modulo 512, and then the length bits. A 64-bit portion is appended to indicate the length of the actual message. MD5 algorithm operates on a 128-bit state which is divided into four 32-bit words (denoted as A, B, C and D) and initialized. Each 512-bit message block is applied in turn to modify the state. The processing of a message block consists of four similar rounds, each of which is composed of 16 similar operations based on a non-linear function F, modular addition, and left rotation. At last, MD5's output is produced by cascading A, B, C and D after the final round.

2.1.2 MD5 Collisions

Hash function is a useful cryptographic tool which should satisfy several requirements to keep it robust and secure on encryption. One of these requirements is collision resistance, i.e., it is hard to compute messages m and m' where $m \neq m'$ and yet $H(m) = H(m')$. A hash function is claimed good if it is extremely difficult to find such existing pairs. MD5 is an encryption-based hash function and suffers a crucial weakness of collision. Finding a collision by a brute force attack requires at most 2^{128} applications of MD5 and 2^{64} by the birthday paradox, since MD5 has 128-bit hash.

Early in 1993, Bert den Boer and Antoon Bosselaers [1] discovered the first pseudo-collision of MD5. In 1996, H. Dobbertin announced a collision of the compression function of MD5 [6]. In 2004 when Xiaoyun Wang and Hongbo Yu presented a collision for MD5 with two input blocks in less than an hour and 5 minutes on a IBM P690 [2]. In 2006, Vlastimil Klima published an algorithm [7] to find a collision within one minute on a single notebook computer, using a "tunneling" method. All these different attacks on MD5 were constructed through multi-block collision method. Tao Xie and Dengguo Feng announced the first single-block MD5 collision (two 64-byte messages with the same MD5 hash) in 2010 [8].

MD5 is recommended to be replaced by some other alternative methods such as SHA-1 and SHA-2, to keep it safe in applications.

2.2 Blowfish algorithm

2.2.1 Algorithm Design

Blowfish is a 64-bit block cipher with a variable-length key (from 32 to 448 bits). It consists of two procedures: key initialization and data encryption phases. The first phase expands a variable user key to 4168/8336-byte sub key arrays, presented by 4-byte element size arrays or 8-byte element size format arrays.

The process of generating subkey arrays (18-entry P and four 256-entry S arrays) also depends on the user key. This enhances the complexity of the user key and subkeys relationship for higher security. Also, in later encryption process, these updated subkeys instead of the user key are used. The F function splits the 32-bit input into four eight-bit quarters as the inputs to the S-boxes. The outputs are added modular 2^{32} and XOR-ed to produce the final 32-bit output. Blowfish adopts Feistel network to iterate a simple encryption function 16 times (rounds). Decryption for Blowfish works similarly, beginning with the ciphertext as input. The difference is that $P_1, P_2, P_3, \dots, P_{18}$ are used in a reverse order.

2.2.2 Blowfish's Deficiency

Blowfish was once believed to resist any attack. However, it is still a symmetric encryption algorithm, i.e., if the secret key is discovered, all of the message can be decrypted easily. Blowfish also suffers the defect of the weak key problem. It means there exists a collision of an S-box. That is, for the key-dependant S-box of Blowfish (S_1), there exist two different bytes of a and a' , such that $S_1(a) = S_1(a')$.

$F(a,b,c,d) = ((S_1(a) + (S_2(b))) \oplus S_3(c)) + S_4(d)$, where \oplus is the bit-wise xor and $+$ is the addition modulo 2^{32} . The 8-bit strings $a, b, c,$ and d are the inputs of F function for each round.

If an attacker knows part of the private key of the F function (or we can say four S-boxes), he/she should be able to recover the plaintext easily with the collision. Serge Vaudenay showed [5] that attacker can recover all the information with 2^{48} chosen plaintexts against a reduced eight-round Blowfish encryption. In addition, attackers do not even need so much some weak F functions. Such attack only needs 2^{32} chosen plaintexts against eight rounds, and 3×2^{51} chosen plaintexts against sixteen rounds. Serge also showed that the possibility of detecting a weak key of one S box is 2^{-15} . Detecting weak keys can be achieved by using 2^{22} chosen plaintexts (on eight rounds). Therefore, it is unsafe to encrypt critical data with single Blowfish encryption algorithm.

2.3 Graphics Processing Unit and CUDA

Graphics Processing Unit (GPU) consists of thousands of processing units with tremendous computational power. The concept of graphics accelerator was proposed between the

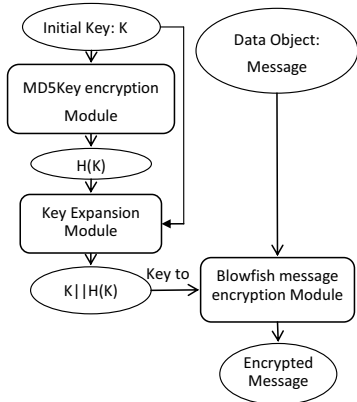


Fig. 1: Message encryption modules in MD5-Blowfish

end of 1970's and the beginning of 1980's. The term of GPU was defined and popularized by NVIDIA in 1999, who marketed the GeForce 256 as "the world's first 'GPU'" [9]. After that, new breakthroughs in GPU technology are announced yearly.

Recently, it was used for data parallel processing. CUDA was developed to ease parallel programming on GPUs [9]. The latest generation of NVIDIA GPU architecture, Fermi, represents a giant step towards bring GPUs into mainstream computing [10]. It provides a plenty of new features, including improved double precision performance, ECC debugging support, true cache hierarchy, more shared memory, faster context switching, and faster atomic operations.

3. Design of MD5-Blowfish

MD5-Blowfish encryption algorithm is designed to enhance the security without much performance loss. This algorithm consists of three major components: an initial key-encryption module, an MD5 key expansion module, and a Blowfish data-encryption module. The initial key is passed into the MD5 key-encryption module and then key expansion module to get the updated encrypted key for the data-encryption module. This process is shown in Fig. 1

3.1 Hybrid Encryption Algorithm and Modules

MD5 Key Encryption Module

The key selected by the user for the whole system is transferred to MD5 key encryption module so that the initial user key is encrypted by MD5. The input user's (private) key may have any length, but the output is fixed with 128 bits long. The reason is that MD5 has no requirement on the input plaintext size, whereas most traditional encryption algorithms such as AES and DES require that the key size be multiple of block size. So it is more convenient for users to select keys.

Key Expansion Module

The encrypted key K' produced by MD5 key encryption module is passed to the key expansion module for further

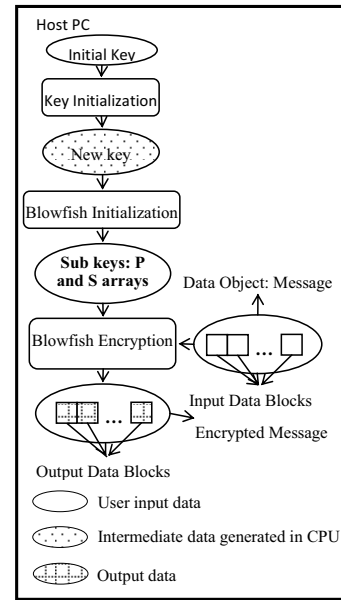


Fig. 2: Implementation on CPU

complexity as shown in Fig. 1. In this module, the encrypted key $H(K)$ is appended to the initial user key K . If the total length is no more than 448 bytes, $K || H(K)$ will be sent to Blowfish message encryption module. Otherwise, only the first 448 bytes will be sent over to ensure correct operations. In addition, it also complicates the whole encryption process for further complexity, i.e., security.

Blowfish Message Encryption Module

The algorithm is exactly same as the Blowfish algorithm except that the key is replaced by the encrypted key from the Key Expansion Module.

It is quite straightforward to implement the hybrid MD5-Blowfish encryption on CPU. The flow chart is shown in Fig. 2. As discussed above, a new key is prepared by the key initialization step, which involves MD5 module and key expansion module. This new key is used as the key to Blowfish encryption in the next step to encrypt the user input data objects. The whole procedure with MD5 key encryption module, key expansion module, and Blowfish message encryption module are executed on CPU. The input plaintext data objects are made up of a number of input data blocks whose number can be calculated as:

$$\#inputdatablocks = \frac{input\ data\ size(bytes)}{128(bytes)}$$

Then these 128-byte input data blocks are sent into the Blowfish encryption module, and encrypted serially on CPU. Accordingly, the same number of 128-byte output data blocks are generated in sequence.

3.2 Parallelized Hybrid MD5-Blowfish

In parallelized MD5-Blowfish algorithm, some technical terms are defined as follows:

Input Data Block: 128 bits of plaintext to be encrypted

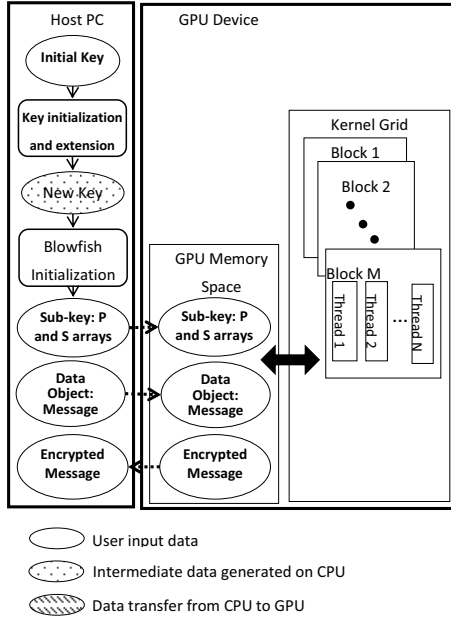


Fig. 3: Implementation on GPU

Output Data Block: 128 bits of ciphertext
 C_P boxes: 16-entry P array produced by CPU
 G_P boxes: 16-entry P array resides in GPU memory space

C_S boxes: four 256-entry S array generated by CPU
 G_S boxes: four 256-entry S array resides in GPU memory space

There are four steps in parallelized hybrid MD5-Blowfish encryption algorithm. First, key initialization and expansion are accomplished by MD5 key encryption module and key expansion module in Host (CPU). Thus the hashed and expanded new key is prepared and ready for the Blowfish initialization procedure. Second, Blowfish initialization procedure generates Sub-keys (C_P boxes and C_S boxes), which is also operated by the host (CPU). Third, Blowfish's Sub-keys (C_P boxes and C_S boxes) and input data objects (a number of input data blocks) are copied into GPU Memory. G_P boxes and G_S boxes are filled with the same value of C_P boxes and C_S boxes. Blowfish's Encryption Module is coded as CUDA kernel functions to encrypt the input message and generate outputs (encrypted message) on GPU. Finally, the encrypted message is copied back to the host and delivered to users as shown in Fig. 3.

CUDA programming on GPU might achieve tremendous speedup for data parallel applications. GPU threads are grouped into blocks which in turn are organized in a grid. Data processing in each GPU thread is shown in Fig. 4. Assume there are M blocks in a grid, and N threads in a block. Each thread in a thread block deals with one 128-bit input data block which contains left and right parts, each for 64 bits of the plaintext. Therefore, totally $M \times N \times 128$ bits of

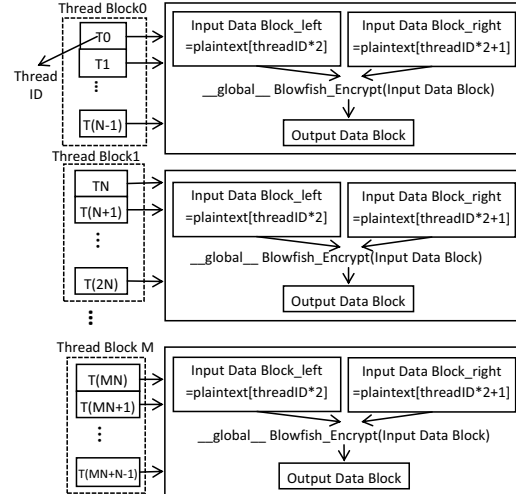


Fig. 4: Data processing in each GPU thread

data is encrypted at one time. There is one finalization step in the end since GPU threads will not finish their computations at the same time. A CUDA synchronization call is issued to ensure all threads have finished their work so that they can be reused safely. At last, threads' outputs are combined together for the final encrypted message.

3.3 Design Issues

The first design issue is to determine the number of blocks and the number of threads in each block. Right now, CUDA compute capacity 2.x defines 1024 as the maximum number of threads per block, 1024 as the maximum x or y dimension of a block, 64 as the maximum z dimension of a block, and 1536 as the maximum number of resident threads per multiprocessor. To thoroughly utilize GPU resources, more threads should be issued simultaneously. The total thread number N_t is related to the input data block number, N_d . User input plaintext is partitioned into numerous 64-bit elements with two for one input data block. Then, the following situations should be considered:

(1) $N_t \geq N_d$. If the plaintext size is smaller than the maximum number of threads in multiprocessor, the thread number should be the same as the input data block number.

(2) $N_t < N_d, N_d \% N_t = 0$. As the plaintext size increases, the input data block number may exceed the maximum number of threads in multiprocessor. Multiple rounds will be required to finish the whole message encryption.

(3) $N_t < N_d, N_d \% N_t \neq 0$. If the number of input data blocks is more than that of threads but not multiple of the thread number, the first $N_d \% N_t$ threads will participate in $(N_d/N_t + 1)$ rounds whereas the others take N_d/N_t rounds. Then the synchronization at the final step becomes more imperative and meaningful. It is easy to imagine some threads with fewer tasks and rounds might finish earlier.

The second design issue is the data storage location. G_P, G_S boxes and input data blocks are copied from host to GPU memory space. In CUDA, the programmer can access multiple GPU memory units which have different access latency, size, operation scope and cache property. Since GPU global and texture memory units allow large data allocation whereas constant memory is limited to 64 KB, G_P box, G_S box and input data blocks are placed in global memory. NVIDIA Fermi also supports L1 and L2 caches with 128-byte cache line. As shown in Table 1, GPU devices we used for experiment have 5,636,554,752 bytes and 5,636,292,608 bytes in global memory, respectively. Since G_P box has $144(18 \times 8)$ bytes and G_S box has $8192(4 \times 256 \times 8)$ bytes, both GPUs still have 5,636,546,416 and 5,536,284,272 bytes left in global memory for the input data blocks. The approximate maximum number of input data blocks is 43 million which is sufficient for many applications.

Finally, GPU program optimization strategies should be considered for performance gains. Compared to CPU and GPU memory bandwidth, the communication channel between CPU and GPU (PCIe) is the major bottleneck. An obvious optimization scheme is to minimize data transfer to and from GPUs. Fortunately, Fermi architecture supports page-locked host memory to achieve this to some extent. The C library function, *malloc()* allocates standard, pageable host memory, while CUDA C supports *cudaHostAlloc()* allocates a block of page-locked host memory, or called pinned memory. An important property of page-locked host memory is that operating systems will guarantee not to page this memory block out to disk. When CUDA kernel functions access the mapped memory, data transfer is implicitly performed without issuing any memory allocation and data transfer operations. Since these memory blocks can never be swapped out to disk, applications need to consider the amount of available physical CPU memory. To go one step further, zero-copy host memory feature can effectively opt out of all the nice properties of virtual memory. Zero-copy host memory is named after the fact that no copy is committed to and from GPU at all. In hybrid MD5-Blowfish algorithm, both page-locked host memory and zero-copy host memory are used to hold P-box, S-box and input data blocks for performance comparison.

GPU can execute kernel functions and transfer data simultaneously by employing multiple CUDA streams. Since input data might be large and data transfer can easily become the performance bottleneck, in hybrid MD5-Blowfish, some threads carry new data into GPU while other threads encrypt the existing one in memory. Memory copy and kernel execution are overlapped to accelerate applications. Obviously multiple GPUs can duplicate aforementioned optimization efforts for linear speedup.

CPU model name	Intel® Xeon (R)	Intel® Xeon (R)
CUDA Drive Version	3.20	3.20
CUDA Runtime Version	3.20	3.20
CUDA Capability Major/Minor Version number	2.0	2.0
Total amount of global memory	5636554752 bytes	5636292608 bytes
Total amount of constant memory	65536 bytes	65536 bytes
Total amount of shared memory per block	49152 bytes	49152 bytes
Total number of registers available per block	32768	32768
Warp size	32768	32768
Maximum number of threads per block	1024×1024×64	1024×1024×64
Maximum sizes of each dimension of a block	65535×65535×1	65535×65535×1
Texture alignment	512 bytes	512 bytes
Clock rate	1.15 GHz	1.15 GHz

Table 1: Specification for GPUs in test

4. Experimental Results

Hybrid MD5-Blowfish is tested for its effectiveness in three ways. First, with data in GPU global memory, the number of blocks and threads are varied for performance comparison. Second, sequential and parallel programs were tested and compared with Intel(R) Xeon(R) X5660 (2.8 GHz, 12,288 KB cache) and two CUDA-capable Fermi GPUs, “Tesla C2070” AND “Quadro 6000” as shown in Table 1. Third, zero-copy, multi-thread, multi-GPU, and GPU work scheduling technologies are adopted to verify performance gains. The total execution time includes MD5 Key initialization and extension as well as Blowfish encryption.

4.1 Kernel Function Configuration

CUDA kernel function is configured by the numbers of blocks and threads per block. In our test, both CUDA-capable Fermi architecture GPUs can hold up to 1024 concurrent threads in a block and 1536 (48 wraps in a multiprocessor and 32 threads per wrap) total threads on one multiprocessor (SM). However, allocated threads are also limited by other SM resources such as shared memory and register file. In total, there could be 14×1536 threads since each GPU has 14 SMs. The number of thread blocks in a grid is usually calculated by the size of the data being processed or the number of cores in the system.

Fig. 5 shows how the block number can affect performance. While the system is configured with 1 block, 2 blocks and 8 blocks in a GPU grid, thread number per block is always set to 1024. The input plaintext size is changed from 32K bytes to 512K bytes. Two steps are involved: step 1 stands for MD5 Key initialization and expansion whereas step 2 is for Blowfish encryption. It is clear that step 1 always takes similar time since it is executed on CPU and also irrelevant to the input plaintext size and block number in CUDA. The total running time increases linearly with the size of input plaintext. When the input plaintext is 32K bytes long, these three configurations

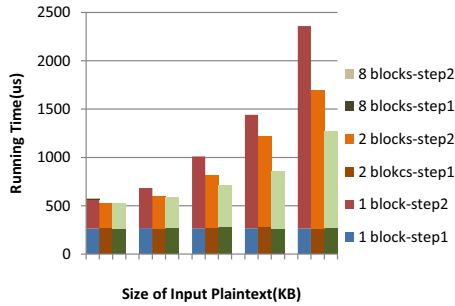


Fig. 5: Influence of the number of blocks

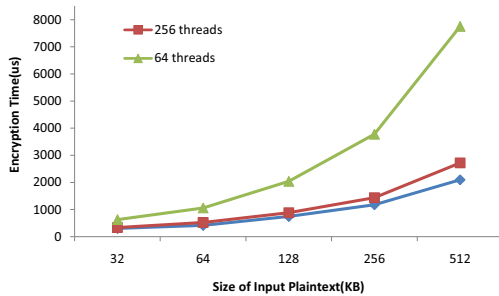


Fig. 6: Influence of the number of thread per block

perform similarly. As the plaintext size is increased to 128K bytes, the 8-block configuration is obviously better than the other two. According to hybrid MD5-Blowfish design, each thread works on two input data blocks (each is 64 bit long). When the input plaintext is not too long, a small number of threads are sufficient to encrypt the plaintext. Too many threads will not improve performance because most of them may remain idle. As the size of input plaintext increases, more threads are involved in the encryption procedure to speed up the entire encryption process.

Fig. 6 demonstrates how the number of threads in a block can affect performance. Only one block is used. The number of threads in this block is changed from 64 to 1024 and the input plaintext size is changed from 32K to 512K bytes. Since the cost in step 1 is fixed, only the cost in step 2, (Blowfish encryption) is considered. Once the thread number reaches 64, the encryption cost increases dramatically. As the size of input plaintext increases, each thread needs to work for multiple rounds. A block with 1024 threads reaches the peak performance since more threads can work encrypt plaintext concurrently.

4.2 CPU vs. GPU Implementations

CPU implementation is based on the flow chart shown in Fig. 2. However, two CPU versions are generated. One is built by default gcc compiler whereas another one is from the optimized gcc compiler by turning on `-O3` command-line option. GPU version is implemented according to Fig. 3. One block is used and 1024 threads are set inside. The input plaintext size is ranged from 8KB to 512KB.

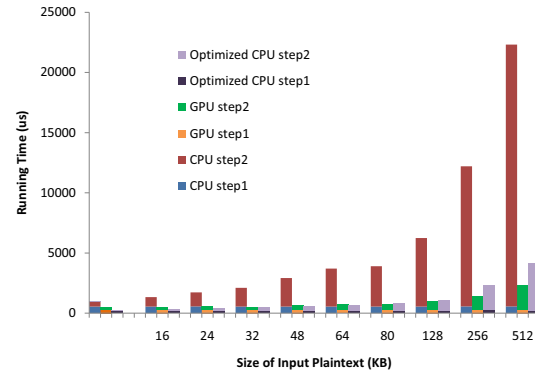


Fig. 7: Performance comparison of CPU and GPU versions

Fig. 7 shows the running time of these three versions. Non-optimized CPU version has the worst performance. Both Step 1 (MD5 Key initialization and expansion) and step 2 (Blowfish encryption) take much longer time than the other two for the same input plaintext. For all of these three versions, step 1 takes almost the same CPU time. NVIDIA's `nvcc` compiler is optimized automatically. So GPU and optimized gcc compiler versions perform better on step 1. When the input plaintext size is smaller (less than 64K), the optimized CPU version outperforms the GPU version. However, as the input size increases, the GPU version becomes much better than those CPU ones.

4.3 GPU Optimizations

With latest CUDA version and Fermi architecture, several advanced features such as multiple streams, zero-copy host memory and multiple GPU technology can improve encryption performance. Fig. 8 demonstrates the costs of step 2 (Blowfish encryption) with four types of kernel configures for possible speedup over the traditional GPU version using global memory. The input plaintext size is increased from 16KB to 512KB.

In the multiple-stream version, the input plaintext is divided into two equal sized parts for two CUDA streams. Then a GPU can execute a kernel function while performing a data copy between the host and GPU. Two memory copies are queued with `cudaMemcpyAsync()` for overlapping of data copy and kernel execution with certain operation orders. When the first memory copy is finished, its data can be used by the kernel function for encryption. At the same time, inside the copy engine, the second part of plaintext text is copied in. In this way, data copy is overlapped with kernel encryption. Proper plaintext partitioning and streams number selection can help pace the kernel execution and data transfer well for better performance.

For multi-GPU version, two GPUs are selected to work simultaneously for more device resources. Each GPU encrypts half of the input plaintext.

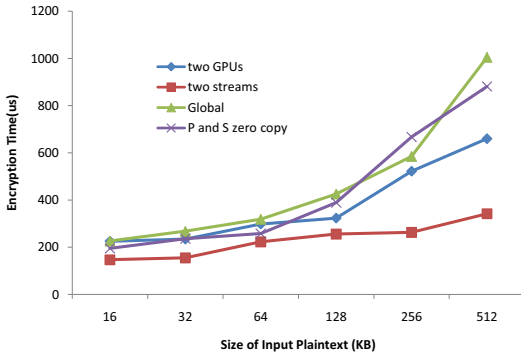


Fig. 8: Encryption time of four optimizations

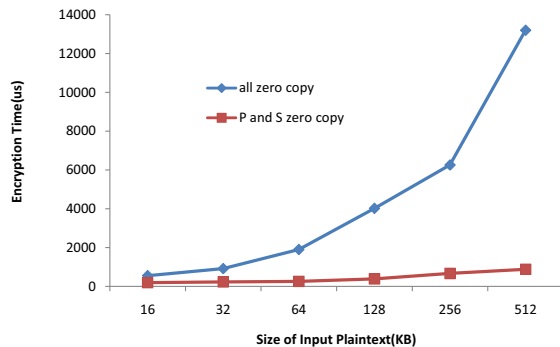


Fig. 9: Encryption time of two zero-copy approaches

At last, zero-copy version can avoid explicit data copies to and from GPU. This maneuver can speed up the encryption since every input data block encryption operation needs P and S boxes. As shown in Fig. 8, this zero-copy version is not always faster than traditional GPU version which needs to transfer data from CPU to GPU. Although data access in zero-copy version might be hidden behind kernel functions, the communication startup overhead in frequent data access for large data might drag down the overall performance. Fig. 9 shows the encryption costs for two zero-copy versions. The “all zero copy” version places input data blocks as well as P and S boxes in zero-copy memory, whereas the “P and S zero copy” only uses zero-copy memory for P and S boxes. The first version runs much slower than the second one. Since the input plaintext needs to be updated after encryption, `cudaThreadSynchronize()` has to be called to synchronize CPU thread with all GPU threads to make sure zero-copy operations are done. However, such synchronization takes too long and degrades the overall encryption performance.

5. Related work

Tremendous enhancements in the field of cryptography have been achieved. Analysis on security and performance of MD5 and Blowfish has been widely discussed and studied, especially after MD5 was turned into a hash cryptography

standard. Several software packages have been developed MD5 and Blowfish. Krishnamurthy G.N and Dr. V. Ramaswamy put forward a modified Blowfish algorithm by modifying F function of the Feistel network [11]. They also proved that this improvement incurs security flaws by comparing it with existing Blowfish algorithm through avalanche effect analysis. They implemented VHDL application to show the different encryption speeds. Their improved algorithm reduces the number of clock cycles required for the execution of Blowfish function by 33%. The overall performance is improved by 14%.

6. Conclusion and future work

Hybrid MD5-Blowfish algorithm is proposed to enhance the security strength and improve the encryption performance over existing MD5 and Blowfish algorithms by merging them. The mixture strategy helps increase the algorithm complexity. Both CPU and GPU versions are developed. Performance analyses for implementations based on different kernel configurations such as different numbers of blocks and numbers of threads in each block as well as latest CUDA and Fermi architecture features such as zero-copy, multi-stream and multi-GPU. Experimental results have indicated impressive performance gains by reducing encryption on GPUs. The future work includes exploring GPU memory hierarchy by placing intermediate data in shared memory, constant memory or texture memory. Also, the fine-tuned data partition and stream generation need further investigation for perfect computation and communication overlapping.

References

- [1] B. den Boer and A. Bosselaers, “Collisions for the compression function of md5,” *Advances in Cryptology EUROCRYPT*, pp. 293–304, 1993.
- [2] X. Wang and H. Yu, “How to break md5 and other hash functions,” *Advances in Cryptology EUROCRYPT*, vol. 3494, pp. 19–35, 2005.
- [3] V. Klima, “Finding md5 collisions on a notebook pc using multi-message modifications,” *Cryptology ePrint Archive Report 2005*, 2005.
- [4] <http://www.schneier.com>, Std.
- [5] S. Vaudenay, “On the weak keys of blowfish,” *FSE1996*, pp. 27–32, 1996.
- [6] H. Dobbertin, “Cryptanalysis of md5 compress,” *EurocrZpt 96*, 1996.
- [7] V. Klima, “Tunnels in hash functions: Md5 collisions within a minute,” *Cryptology ePrint Archive Report*, 2006.
- [8] <http://eprint.iacr.org/2010/643>, Std.
- [9] J. Sanders and E. Kandrot, *CUDA by example : An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 2010.
- [10] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2010.
- [11] K. G.N, D. V. Ramaswamy, L. G.H, and A. M.E, “Performance enhancement of blowfish and cast-128 algorithms and security analysis of improved blowfish algorithm using avalanche effect,” *IJCSNS International Journal of Computer Science and Network Security*, vol. 8, 2008.

Multi-GPU Load Balancing for In-situ Visualization

R. Hagan and Y. Cao

Department of Computer Science, Virginia Tech, Blacksburg, VA, USA

Abstract—*Real-time visualization is an important tool for immediately inspecting results for scientific simulations. Graphics Processing Units (GPUs) as commodity computing devices offer massive parallelism that can greatly improve performance for data-parallel applications. However, a single GPU provides limited support which is only suitable for smaller scale simulations. Multi-GPU computing, on the other hand, allows concurrent computation of simulation and rendering carried out on separate GPUs. However, use of multiple GPUs can introduce workload imbalance that decreases utilization and performance. This work proposes load balancing for in-situ visualization for multiple GPUs on a single system. We demonstrate the effectiveness of the load balancing method with an N-body simulation and a ray tracing visualization by varying input size, supersampling, and simulation parameters. Our results show that the load balancing method can accurately predict the optimal workload balance between simulation and ray tracing to significantly improve performance.*

Keywords: Multi-GPU Computing, Load Balancing, In-situ Visualization, N-body Simulation, Ray Tracing

1. Introduction

GPU computing offers massively parallel processing that can greatly accelerate a variety of data parallel applications. Use of multiple GPUs can lead to even greater performance gains by overlapping computations by executing multiple tasks on different GPUs. This provides an opportunity for handling larger scale problems that a single GPU cannot process in real-time. The resulting increase in runtime speeds can allow for real-time navigation and interaction, which can lead to a much more effective visualization experience. By designing effective algorithms to run on multiple GPUs, a considerable improvement in computational power can be realized. Effective load balancing can greatly increase utilization and performance in a multi-GPU environment by distributing workloads equally.

These properties make multiple GPUs suitable for in-situ visualization applications that use the GPU for concurrent simulation and rendering for interactive visualization. N-body simulation is one such application that involves computation of the interaction among a group of bodies. The N-body problem can be solved by computing the force of all bodies on each other. This problem is used in many domains, including biomolecular and physics applications.

As in the work of [1], the gravitational N-body problem can be expressed as

$$F_i = Gm_i \sum_{1 \leq j \leq N, j \neq i} \frac{m_j r_{ij}}{\|r_{ij}\|^3} \quad (1)$$

where F_i is the computed force for body i , m_i is the mass of body i , r_{ij} is the vector from body i to j , and G is the gravitational constant.

In a molecular simulation, an N-body simulation algorithm can be used to compute the interaction of each atom in the molecule. This application can benefit from use of multiple GPUs to both compute new frames of simulation and render these new frames in parallel. Computation of simulation with rendering in real-time allows for interactive update in visualization applications. This can result in a smooth interaction experience with use of increased processing power to accelerate computing.

While multiple GPUs can offer a large performance gain in visualization applications, many challenges exist in scheduling multiple tasks. Load imbalance can lead to underutilization of available resources and reduced performance in the visualization. Multi-GPU computing can therefore benefit from a method for load balancing to improve workload distribution. Load balancing needs to account for performance in order to maximize use of available resources. Use of the load balancing method in an N-body simulation accounts for workload differences between simulation and rendering to maintain more equal workload distribution. Visualization algorithms such as ray tracing can be computationally expensive, so specific techniques to distribute and load balance rendering workloads can offer considerable performance gains for visualization applications. Taking advantage of concurrent computations of visualizations with simulation can lead to a significant performance improvement to maintain interactivity in these applications.

While load balancing can improve performance when using multiple GPUs, several factors need to be accounted for in visualization. The cost of simulation and rendering can depend on the chosen algorithms for each. The input data size may differ for applications, which can have a varying effect on simulation and rendering time. Accuracy of simulation can be improved by using more accurate techniques or by decreasing the timestep in simulation. In visualization, image quality is important to produce a better result for interactive viewing. Ray tracing is a rendering method that can produce realistic results on the GPU for visualization.

Supersampling can further improve image quality by using multiple samples per pixel that can decrease aliasing. Use of ray tracing with supersampling can greatly improve the results in interactive visualization but significantly increases computations that can be accelerated through use of multiple GPUs. Supersampling and improving the accuracy of simulation can vary the cost of computations, which will require adjusting load balancing for multi-GPU processing. Our method addresses this issue to improve performance with multi-GPU visualization.

Due to the significant gains possible with use of multiple GPUs, we implement load balancing for multi-GPU visualization applications. Our work provides several contributions, including:

- Acceleration of an N-body simulation and ray tracing application using multiple GPUs
- Performance analysis of workload variation based on multiple input parameters
- A load balancing method to predict optimal workload distribution and significantly improve performance

2. Related Work

There have been several related areas of previous research, including multi-GPU computing and visualization using the GPU.

Previous work in multi-GPU visualization has included several applications that use multiple GPUs for rendering. Fogal et al. present a system for visualizing volume datasets on a GPU cluster [2]. However, their work could benefit from additional load balancing between GPU tasks that could provide more flexible and effective workload distribution for simulation and rendering. Monfort et al. present an analysis of split frame and alternate frame with multiple GPUs for a game engine [3]. They present an analysis of load balancing for a combined rendering mode to improve utilization of multiple GPUs. Binotto et al. present work in load balancing in a CFD simulation application [4]. They use both an initial static analysis followed by adaptive load balancing based on various factors including performance results. While these previous works present load balancing techniques, they do not focus on load balancing for in-situ visualization based on rendering and simulation tasks. We present a performance model and load balancing technique for simulation and rendering that allows improved load balancing and accounts for the pipelining process necessary in a multi-GPU environment.

Other work has focused on streaming for out-of-core rendering. Gobbetti et al. present Far Voxels, a visualization framework for out-of-core rendering of large datasets using level-of-detail and visibility culling [5]. Crassin et al. present GigaVoxels, an out-of-core rendering framework for volume rendering of massive datasets using a view-dependent data representation [6]. While our load balancing method also uses multiple GPUs and similar pipelining to visualize

datasets, we focus specifically on load balancing for in-situ visualization using ray tracing.

Several other frameworks have been proposed that use multiple GPUs for general-purpose computations. Harmony presents a framework that dynamically schedules kernels [7]. Merge provides a framework heterogeneous scheduling that exposes a map-reduce interface [8]. DCGN is another framework that allows for dynamic communication with a message passing API [9]. However, these works focus on providing a general framework not specific to visualization and could benefit from additional tools for load balancing. We employ similar techniques to improve multi-GPU performance, but we provide improved load balancing in a visualization and simulation application.

Other previous work has focused on GPU computing in molecular dynamics applications, relating to the N-body simulation and visualization used in our work. Past work has included Amber, a molecular dynamics software package that offers tools for molecular simulation [10]. This simulation can be used to compute the change in atoms over time due to an N-body simulation. Other research in molecular dynamics has included work by Anandakrishnan et al. to use an N-body simulation to compute the interaction of atoms in a molecule [11]. Humphrey et al. present Visual Molecular Dynamics (VMD), a software package for visualization of molecular datasets [12]. However, VMD provides primarily off-line rendering that does not simulate and render each frame interactively to allow for real-time user interaction. Stone et al. present work that computes molecular dynamics simulations on multi-core CPUs and GPUs [13]. Furthermore, they visualize molecular orbitals in an interactive rendering in VMD. We also apply our work to an N-body simulation, but we focus on load balancing using multiple GPUs for both simulation and rendering while their work focuses on data parallel algorithms on single GPUs. Chen et al. present work in multi-GPU load balancing applied to molecular dynamics that uses dynamic load balancing with a task queue [14]. Their framework focuses on fine-grained load balancing usable within a single GPU, while our work focuses on coarse-grained task scheduling among GPUs for both simulation and visualization. While there has been considerable work in visualization and multi-GPU computing, we focus on a method for load balancing between simulation and rendering to improve utilization in the pipelined memory model useful for in-situ visualization.

3. Methods

Our approach addresses the issue of workload imbalance for in-situ visualization applications. We will first describe the problems in this application area, and then we present our load balancing method for solving these issues.

3.1 Multi-GPU Architecture

In comparison with a single GPU, a multi-GPU implementation has several advantages. Most notably, multiple GPUs can overlap concurrent computations on several GPUs at once. However, unlike a single GPU, memory transfers are required to ensure that a GPU has the required data. Figure 1 shows the multi-GPU configuration used with our application. It identifies how multiple GPUs can be used for overlapping computation between simulation and rendering, while host memory is used to transfer results among GPUs. For in-situ visualization applications, the simulation data must be transferred to rendering tasks to render the resulting image. This data is first transferred to the host and then transferred to the recipient GPU. This creates a pipelined model of execution where multiple GPUs can process data concurrently but must transfer data through host memory.

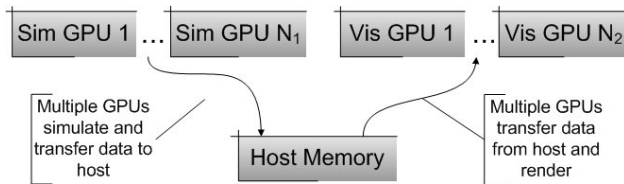


Fig. 1: Diagram of multi-GPU configuration: "Sim" refers to simulation and "Vis" refers to visualization

3.2 Multi-GPU Workload Imbalance

While use of multiple GPUs can greatly increase the available processing power and improve performance by overlapping computation, use of multiple GPUs introduces issues of workload distribution among processors. This workload imbalance results from the synchronization necessary through host memory. Each task either sends or receives data through a buffer. However, use of a single buffer would require simulation tasks to wait for rendering tasks to read this data before writing the next frame. This can result in significant idle times that can decrease utilization and performance. Multiple buffers can allow one task to read or write data to multiple buffers before having to wait for other tasks to process the data as shown in Figure 2. Thus, having multiple buffers can improve load balance at the cost of additional memory.

The amount of host memory is finite, however, which requires the tasks to eventually wait if workload imbalance is significant. If simulation of a single frame requires less time than rendering, host memory eventually becomes full, which requires simulation to wait for rendering. When rendering of a single frame takes less time than simulation, the buffers in host memory become increasingly empty, which requires rendering tasks to wait for simulation. Figure 3 shows the case where performance can decrease due to improper workload distribution. Since rendering GPUs need to read simulation data before simulation can overwrite it, idle

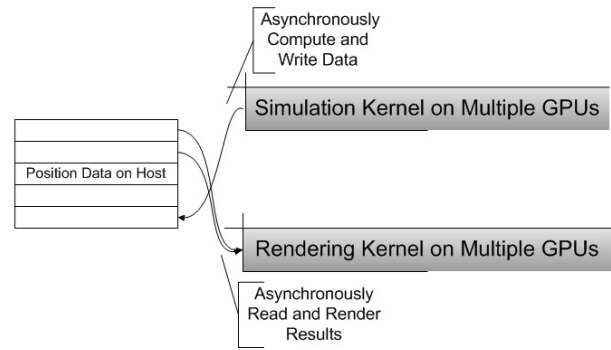


Fig. 2: Memory transfers between host and GPU memory

time can be introduced if simulation time is shorter than rendering.

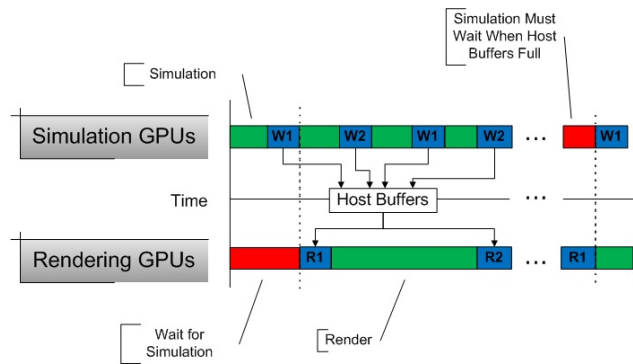


Fig. 3: Simulation tasks must wait for ray tracing to read results. "W1" refers to writing data to the first buffer in host memory from a GPU, while "R2" refers to reading data from the second buffer in host memory

Accounting for workload imbalance can eliminate this idle time by distributing work equally among available processors as shown in Figure 4. Load balance among tasks allows tasks to send and receive data at an equal rate and thus improve utilization. However, this requires formulating a method for load balancing. Varying workloads for tasks creates issues for the problem of load balancing. For example, the type of rendering technique or number of samples in supersampling can change the workload and introduce additional idle time. Factors of simulation such as accuracy or type of simulation could also affect runtime, resulting in a different optimal workload balance as well. These various issues demonstrate the important need for load balancing for in-situ visualization. Given an initial set of characteristics for rendering and simulation for a specific visualization, finding the optimal load balance can reduce idle time and improve performance.

3.3 Load Balancing

The use of our multi-GPU implementation allows for load balancing techniques for in-situ visualization. Our test

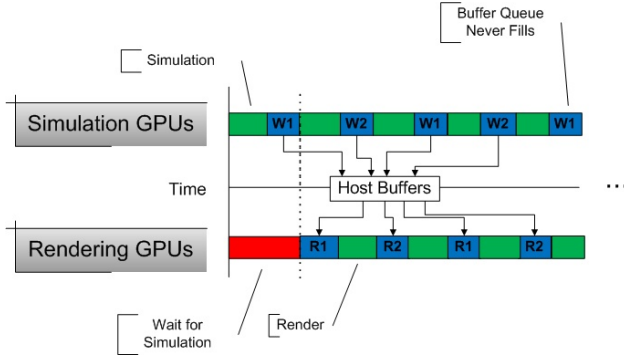


Fig. 4: Use of load balancing reduces idle time

application uses a gravitational N-body simulation based on the method of [1] to compute interactions of particles, while ray tracing renders the results. Particle position data is transferred through the host using multiple buffers in order to pipeline position data between simulation and rendering. Since simulation and rendering may have different amounts of workload, it is important to address the possibility of workload imbalance between the tasks.

In this application load balancing requires partitioning the dataset in order to distribute work to GPUs. Two types of work partitioning are possible with the application: *inter-frame* and *intra-frame*. Inter-frame partitioning involves distributing complete frames of data in order to achieve load balancing. Ray tracing in this implementation uses inter-frame partitioning to render entire frames in order to avoid communication in combining results and improve performance. Intra-frame partitioning distributes parts of a single frame to GPUs for processing. The N-body simulation utilizes intra-frame partitioning by having each GPU update only a subset of the particles for a single frame of data. Since each frame of simulation requires previous data, simulation cannot be computed out of order. Thus, multiple GPUs can only accelerate simulation by having each GPU update a subset of the dataset. Intra-frame partitioning for simulation is therefore necessary to apply load balancing. While this requires communication to combine the results for each frame, the computation can be distributed among multiple GPUs. Thus, this load balancing method uses groups of GPUs to compute frames of data for simulation while rendering has single GPUs separately compute rendering results for consecutive frames. The partitioning of both rendering and simulation tasks allow for load balancing in the visualization application. As more processors are dedicated to simulation, fewer are dedicated to rendering consecutive frames. The total visualization time for a frame is used to determine the optimal load balance between rendering and simulation among the available processors.

To address the issue of workload imbalance, we present a load balancing method to achieve the optimal distribution of work to improve performance. We present a

three-dimensional parameter matrix M that can be used to determine the appropriate balance for the visualization application. The input dimensions of M include the number of samples for supersampling, the number of iterations for simulation, and the input size, while the associated output values are performance times for these configurations. Our method first collects performance results for this matrix and then computes the desired workload balance for a new set of input parameters. Thus, we find the solution to the function:

$$f(i, s, p) = g \quad (2)$$

where f is the function to compute optimal workload distribution, i is the number of iterations for simulation, s is the number of samples for supersampling, p is the number of particles in the simulation, and g is the number of GPUs allocated for rendering versus simulation. The predicted optimal load balance is computed based on previous results through trilinear interpolation. Given known optimal workload distributions for sets of input parameters, our model predicts the optimal load balancing result g for a new set of input parameters:

$$\begin{aligned} L_{isp} = & L_{000}(1-i)(1-s)(1-p) + L_{100}i(1-s)(1-p) \\ & + L_{010}(1-i)s(1-p) + L_{001}(1-i)(1-s)p \\ & + L_{101}i(1-s)p + L_{011}(1-i)sp \\ & + L_{110}is(1-p) + L_{111}isp \end{aligned} \quad (3)$$

where L is the optimal load balance, i is the number of simulation iterations, s is the number of samples for ray tracing, and p is the number of particles. Here, i , s , and p are normalized to the range $[0, 1]$ for interpolation, and the result g is rounded to the nearest integer. This result gives a prediction for the optimal load balance for a given set of input parameter values.

4. Results

The multi-GPU load balancing method was tested with an N-body simulation and ray tracing of thousands of spheres. All tests were done on a single computer with eight GTX 295 graphics cards.

The final result of the visualization and the differences in supersampling can be seen in Figure 6. Aliasing artifacts due to inadequate sampling can be seen in the image on the left with one sample per pixel. Using sixteen samples per pixel in a random fashion, however, significantly improves the results.

While supersampling improves the quality of the final image, it comes at a performance cost as shown in Table 1. The increase in execution time for a greater number of samples for supersampling is linear. Thus, the tradeoff between performance and image quality must be considered when

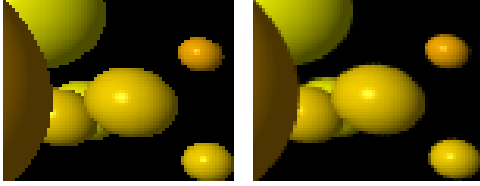


Fig. 5: Comparison of single sample (left) and 16 sample randomized supersampling (right)

Table 1: GPU execution time (ms) for ray tracing based on number of samples for supersampling for 1000 particles

1 sample	4 samples	8 samples	12 samples	16 samples
46.082 ms	163.82 ms	316.56 ms	469.39 ms	621.60 ms

choosing an appropriate number of samples for supersampling.

Table 2 shows the performance time for simulation when performing multiple iterations with a smaller timestep. The performance of simulation shows a linear increase in time with an increase in number of iterations. While a smaller timestep provides more accurate simulations, it introduces additional computations for each frame. Thus, using a smaller timestep but increasing the number of iterations leads to an increase in performance. Table 3 shows the percent difference in positions of simulation from 12000 iteration simulation, which uses the smallest timestep. Each simulation is carried out over the same total time, with a smaller timestep for simulations run for more iterations. With a smaller timestep, the accuracy of the simulation is improved due to the finer granularity used for integration in the N-body simulation.

Table 4 shows a linear decrease in the execution time for simulation when partitioning the dataset to simulate on multiple GPUs. Due to slight constant overhead of launching the kernel, etc., six GPUs gain a slightly less than six times speedup over use of one GPU.

Ray tracing has a longer execution time than simulation for smaller dataset sizes. Simulation takes less time for smaller datasets, but with an increased number of simulation iterations this cost can exceed that of ray tracing with fewer samples. These differences in workload affect the final optimal load balance.

Table 2: GPU execution time for simulation based on number of iterations

20 iterations	40 iterations	60 iterations	80 iterations
31.87 ms	62.56 ms	92.70 ms	122.86 ms

Table 3: Percent difference in positions of simulation from 12000 iteration simulation

Iterations	2000	4000	6000	8000	10000	12000
Percent	58.07	35.37	26.87	21.81	14.94	0.00

Table 4: Execution time for simulation based on number of GPUs used

1 GPU	2 GPUs	3 GPUs	4 GPUs	5 GPUs	6 GPUs
31.87 ms	16.69 ms	11.52 ms	9.32 ms	7.58 ms	6.44 ms

4.1 Workload Characteristics

The multiple input parameters for this application result in many possibilities for workloads. These varying workloads can introduce a performance penalty if not accounted for in distribution of work. Figure 7 shows the trends for performance times for different workloads (number of ray tracing tasks) with varying dataset sizes with 16 sample ray tracing and 80 iteration simulation. The cost of simulation increases more as dataset size increases due to the nature of the N-body simulation, while ray tracing scales linearly with dataset size. This causes the overall performance to be increasingly limited by simulation time for larger datasets.

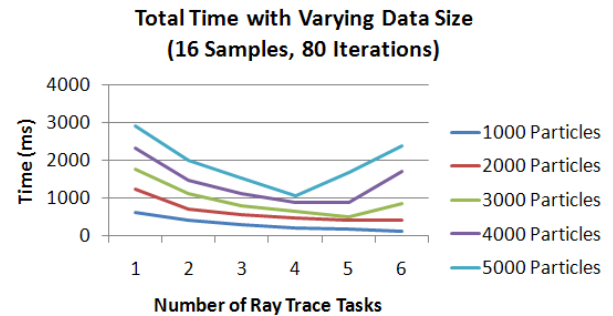


Fig. 6: Performance time for various input sizes with 16 samples, 80 iterations

Figure 8 shows performance for different input sizes with a varying workload distribution for four sample ray tracing and 80 iterations for simulation. This graph shows that allocating more GPUs for simulation when the number of samples is low can result in performance gain. The difference in the trend from Figure 7 also demonstrates that different input parameters can lead to significantly different optimal workload distributions that requires load balancing.

4.2 Load Balancing

Figure 9 shows a trend of optimal load balance based on the number of iterations for simulation. As shown, increasing the number of iterations requires a greater number of GPUs dedicated to simulation to achieve optimal load balance. With the fewest iterations for simulation, the majority of GPUs should be allocated for ray tracing due to the greater cost of ray tracing.

Increasing the number of samples for supersampling increases the cost of ray tracing and also impacts the load balancing scheme. Figure 10 shows that increasing the number of samples for supersampling results in need of

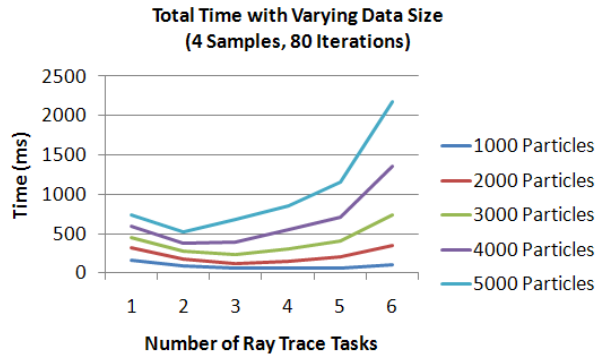


Fig. 7: Performance time for various input sizes with four samples, 80 iterations

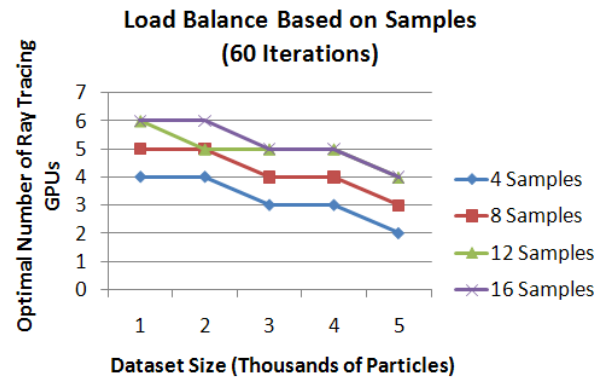


Fig. 9: Load balancing for various numbers of samples for ray tracing with changing dataset size

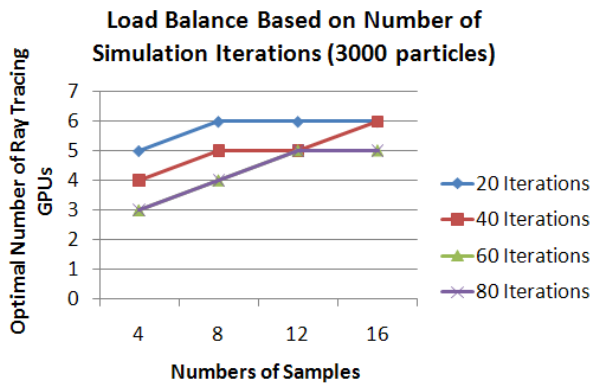


Fig. 8: Load balancing for various simulation iterations with four samples, 3000 particles

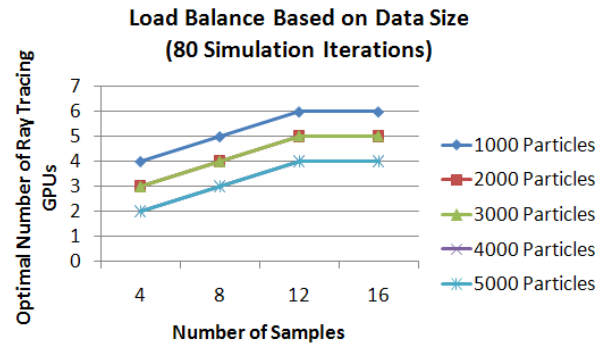


Fig. 10: Load balancing for various dataset sizes with 80 simulation iterations

additional ray tracing tasks to improve workload balance. A larger dataset size requires fewer GPUs for ray tracing due to the smaller increase in cost of ray tracing with larger datasets.

Figure 11 shows the trend for varying dataset size and number of samples with a constant simulation. With a larger dataset size, simulation becomes increasingly expensive while ray tracing cost increases at a linear rate. Therefore, it becomes necessary to compute simulation on an increasing number of GPUs with larger datasets to maintain workload balance.

These results demonstrate that significant workload imbalance can be introduced based on differing workloads of rendering and simulation. Each configuration leads to a different optimal load balancing configuration. A load balancing method must be able to account for these varying trends in order to achieve effective performance.

4.3 Performance Model

We now present a summary of the results of applying our load balancing method. Table 5 shows the percent error

in our proposed prediction model when compared to the actual optimal load balanced configuration. The performance model was tested by computing the average percent error for 40 values for varying one dimension (simulation iterations), 20 for varying two dimensions (iterations and samples), and 12 for varying three dimensions (iterations, samples, and data size). These results show that the average percent difference is below 5 percent for two and three dimensions, and below 10 percent for all categories. Interpolation with fewer dimensions yields a larger error due to discretization error with selection of number of GPUs for load balancing. Using a larger number of data points with more dimensions in interpolation decreases this discretization error.

The performance of our load balancing method was also compared against the worst and average case workload distribution. Figure 12 shows the speedup of using the load

Table 5: Percent error in load balancing model for a varying number of dimensions

1 dimension	2 dimensions	3 dimensions
9.88%	2.67%	1.67%

balancing method. These results show that by using the model, a significant speedup can consistently be achieved with different parameter configurations.

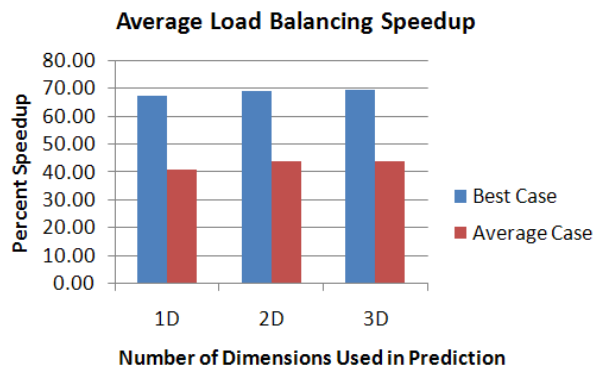


Fig. 11: Load balancing speedup over the average and worst cases

5. Conclusions and Future Work

We have proposed a multi-GPU load balancing solution for in-situ visualization. We have presented an analysis of the workload properties and load balancing results in an N-body simulation and ray tracing visualization. Our results show that workloads can vary greatly for different sets of input parameters, which demonstrates the need for load balancing in multi-GPU computing. Our multi-GPU implementation demonstrates the use of intra- and inter-frame task partitioning for scheduling of GPU tasks to allow the use of load balancing. The results of our tests show that the load balancing method can accurately predict optimal workload balance to significantly improve performance by increasing utilization of available resources.

This work could be extended in multiple ways in future work. Our load balancing approaches could be extended to additional visualization applications, where other rendering and simulation methods with varying workloads could also be addressed. Different performance models may be useful for other applications as well. The pipelining model used in this application would also be useful for out-of-core rendering of massive models.

6. Acknowledgements

This work is funded by Air Force Research Laboratory Munitions Directorate, FA8651-11-1-0001, titled "Unified High-Performance Computing and Visualization Framework on GPU to Support MAV Airframe Research." I also thank Dr. Eli Tilevich for support and discussion on the project.

References

- [1] J. P. L. Nyland, M. Harris, "Fast n-body simulation with cuda," *GPU Gems 3*, pp. 677–695, 2007.
- [2] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, and P. Hatcher, "Large data visualization on distributed memory multi-gpu clusters," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 57–66. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1921479.1921489>
- [3] J. R. Monfort and M. Grossman, "Scaling of 3d game engine workloads on modern multi-gpu systems," in *Proceedings of the Conference on High Performance Graphics 2009*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 37–46. [Online]. Available: <http://doi.acm.org/10.1145/1572769.1572776>
- [4] A. Binotto, C. Pereira, and D. Fellner, "Towards dynamic reconfigurable load-balancing for hybrid desktop platforms," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 04 2010, pp. 1–4.
- [5] E. Gobetti and F. Marton, "Far voxels: a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms," in *ACM SIGGRAPH 2005 Papers*, ser. SIGGRAPH '05. New York, NY, USA: ACM, 2005, pp. 878–885. [Online]. Available: <http://doi.acm.org/10.1145/1186822.1073277>
- [6] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, "Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering," in *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, ser. I3D '09. New York, NY, USA: ACM, 2009, pp. 15–22. [Online]. Available: <http://doi.acm.org/10.1145/1507149.1507152>
- [7] G. F. Diamos and S. Yalamanchili, "Harmony: an execution model and runtime for heterogeneous many core systems," in *Proceedings of the 17th international symposium on High performance distributed computing*, ser. HPDC '08. New York, NY, USA: ACM, 2008, pp. 197–200. [Online]. Available: <http://doi.acm.org/10.1145/1383422.1383447>
- [8] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," *SIGPLAN Not.*, vol. 43, pp. 287–296, March 2008. [Online]. Available: <http://doi.acm.org/10.1145/1353536.1346318>
- [9] J. A. Stuart and J. D. Owens, "Message passing on data-parallel architectures," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.
- [10] D. A. Pearlman, D. A. Case, J. W. Caldwell, W. S. Ross, T. E. Cheatham, S. DeBolt, D. Ferguson, G. Seibel, and P. Kollman, "Amber, a package of computer programs for applying molecular mechanics, normal mode analysis, molecular dynamics and free energy calculations to simulate the structural and energetic properties of molecules," *Computer Physics Communications*, vol. 91, no. 1-3, pp. 1–41, 1995. [Online]. Available: <http://www.sciencedirect.com/science/article/B6TJ5-4037S49-D/2/0df1c6e2cbc422472f7498040a749b20>
- [11] R. Anandakrishnan and A. V. Onufriev, "An n log n approximation based on the natural organization of biomolecules for speeding up the computation of long range interactions," *Journal of Computational Chemistry*, vol. 31, no. 4, pp. 691–706, 2010. [Online]. Available: <http://dx.doi.org/10.1002/jcc.21357>
- [12] W. Humphrey, A. Dalke, and K. Schulten, "VMD – Visual Molecular Dynamics," *Journal of Molecular Graphics*, vol. 14, pp. 33–38, 1996.
- [13] J. E. Stone, J. Saam, D. J. Hardy, K. L. Vandivort, W.-m. W. Hwu, and K. Schulten, "High performance computation and interactive display of molecular orbitals on gpus and multi-core cpus," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 9–18. [Online]. Available: <http://doi.acm.org/10.1145/1513895.1513897>
- [14] L. Chen, O. Villa, S. Krishnamoorthy, and G. Gao, "Dynamic load balancing on single- and multi-gpu systems," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 04 2010, pp. 1–12.

Designing a Parallel Collaborative SAT Solver

Pascal Vander-Swalmen¹, Gilles Dequen², Michaël Krajecki³

¹Université de Versailles Saint-Quentin-en-Yvelines (email address: prism.uvsq.fr)

²Université de Picardie Jules Verne (email address: u-picardie.fr)

³Université de Reims Champagne-Ardenne (email address: univ-reims.fr)

Abstract—*The combinatorial optimization problems are difficult to set up within a parallel context: a search-space is built during the computation and is explored in an irregular way. Moreover, the size of the search-space grows exponentially according to the size of the problem. Since the current processor industry is multiplying the number of cores in their product instead of increasing their frequency, setting up a parallel scheme to combinatorial problem remains needed. In this paper we focus on the SAT problem, which is central in theory of the Computer Science. We show different ways to solve SAT in parallel and we explain the advantages and the shortages of them. Then we explain how to design a parallel SAT solver that is able to keep the very efficient sequential techniques while parallelizing the work among the available cores. Obviously, the key-feature for the SAT community is to parallelize a sequential SAT solver since last sequential improvements are embedded in it. To answer these problems we propose a parallel SAT solver called MTSS (for Multi-Threaded SAT Solver).*

Keywords: Parallelism, Combinatorial Optimization, Satisfiability

1. Introduction

The Satisfiability Problem (short for SAT) is a well-known \mathcal{NP} -complete problem [9] and is a core problem in mathematical logic and computing theory. The interest in studying SAT has grown significantly over the last years because of its conceptual simplicity and its ability to express a large set of various problems. To date, it remains a central problem in artificial intelligence, logic and computational complexity theory. This leads to propose a new class of algorithms [7]. Within a more practical framework, a lot of works highlight SAT implications in “real world” problems as diverse as Planning [17], Model Checking [6], Cryptography [20], VLSI design, ...

In recent years, several improvements, dedicated to the original backtrack-search DLL procedure [10], about splitting variable choice [11], [2], restart strategies [3], preprocessing [18], fast unit-propagation and more generally logical simplification techniques [1] have allowed SAT solvers to be very efficient in solving huge problems from industrial areas¹. Basically, the sequential solvers are very efficient and

integrate a lot of techniques which are designed and tuned to achieve tremendous performances. Hence, the complexity of the different pruning and speeding-up techniques implementation coupled with their practical sequential efficiency lead the parallel design of state-of-the-art sequential SAT solvers to be very difficult and time consuming unless to obtain a weak parallel efficiency. Tackling this issue is a challenging problem.

In section 2, we explain several ways to parallelize a SAT solver according to its characteristics. The section 2.3 gives a non exhaustive list of some important parallel SAT solvers, especially some of them quite recent for multi-core computers. In section 3, a description of a parallel collaborative SAT solver that is able to keep all the sequential techniques is given. In section 4, the solver MTSS is presented. It follows the rules of the collaborative parallel solver and is able to parallelize sequential SAT solvers. Section 5 presents results obtained when parallelizing sequential SAT solvers on multi-core computers (the parallelized solvers are March [15], Kcnfs [11] and Minisat [22]). Some future works and perspectives are presented as a conclusion in section 6.

2. Parallel SAT solving

2.1 SAT

Let $\mathcal{V} = \{v_1, v_2, \dots, v_n\}$ be a set of *boolean variables*. A *literal* is the signed form of a boolean variable. Let denote respectively v and \bar{v} the positive and negative literals associated to the variable v . A *CNF-Formula* \mathcal{F} is a conjunction of *clauses*, where a clause is a disjunction of literals. An *interpretation* of \mathcal{F} is an assignment of truth values $\{true, false\}$ to \mathcal{V} . The literal v (resp. \bar{v}) is satisfied if the variable v has the value TRUE (resp. FALSE). A clause is satisfied if at least one of its literals is satisfied. Finally, \mathcal{F} is *satisfiable* if it exists an interpretation where all the clauses are satisfied (this interpretation is then a *solution*). The SAT problem is to decide if there exists a solution for \mathcal{F} , if not, \mathcal{F} is *unsatisfiable*.

The SAT community mainly works on two types of problems. On the one hand, the randomly generated formulas that follow a generation model which consists in uniformly, independently and without replacement choosing m clauses to n variables among the $2^k \binom{n}{k}$ non-trivial clauses with k distinct and non-tautological literals. The main studies on this field are intended to the random 3-SAT problem (i.e.

¹<http://www.satcompetition.org>

$k = 3$). A phase transition phenomena correlated to the ratio $r = \frac{m}{n}$ has been identified in [19]. Thus, it is also experimentally shown that for low range of r , generated formulas tend to be SAT whereas for high range of r generated formulas tend to be UNSAT. Moreover, around specific range of r , the probability to conclude SAT is about 0.5 and the associated solving time grows up exponentially. To date, the most difficult formulas proposed and solved by the SAT community are random 3-SAT formulas with 700 variables and 2975 clauses [12]. In the following, we focus our experimental comparison on solving random 3-SAT formulas generated with $r = 4.25$ which is the pick of difficulty area. On the other hand, within a more practical point of view SAT remains an easy way to express various problems. This leads to have a widespread use of SAT in many industrial applications. Industrial formulas often have several thousands of clauses and variables but are easier to solve in terms of practical complexity. Due to their huge size, lazy complete solving proposed by the CDCL [25] procedure (“*Conflict Driven Clause Learning*”) is considered being the most efficient. As for DLL, from which it derives, CDCL is a complete approach and builds a binary search-tree. It has some specific treatments due to its lazy behavior like the use of a branching heuristic based on an analysis of the variables occurrences in the encountered conflicts. Analyzing a conflict allows, thanks to a clause learning, to avoid it in the future process. Finally, CDCL-based solvers periodically erase the search-tree and then restart keeping learnt informations. The objective is to develop a smaller tree. In the following, we also propose experimentations on industrial benchmarks from the SAT competitions.

Two classes of SAT solving techniques are commonly used by the community.

- *Complete* approaches guarantee an answer in a finite but exponential runtime. These methods are mainly based on the DLL [10] algorithm which consists in a systematic search in the search-space of truth assignments thanks to a binary search-tree. The Figure 1 shows the skeleton of this procedure with its recursive enumerative feature where $\mathcal{F} \setminus x$ (resp. $\mathcal{F} \setminus \bar{x}$) is \mathcal{F} logically simplified with inference rules from $x = True$ (resp. $x = False$).
- *Incomplete* SAT solving (or general Constraint Satisfaction Problem) methods are those that cannot guarantee an answer in a finite runtime. The relaxation of this guarantee leads these methods to practically behave as polynomial algorithms. Hence, depending on their success rate, they are able to answer more quickly than complete techniques. In practice, most such methods are dedicated to satisfiable formulas and are unfortunately not able to prove the unsatisfiability. Moreover, they are intended to specific classes of problems such as randomly generated ones. Among the incomplete approaches to SAT solving, one of the most efficient

is based on GSAT and Walksat algorithms [21] which can be briefly describe as greedy and noisy searches into the search-space. The reader should refer to [5] for more details.

This work, that aims at proposing a new parallel solver named MTSS, focuses on SAT solving based on DLL procedure and is dedicated to solve randomly generated formulas and to a lesser extent the industrial formulas thanks to the parallelization of several instances of a sequential and industrial SAT solver.

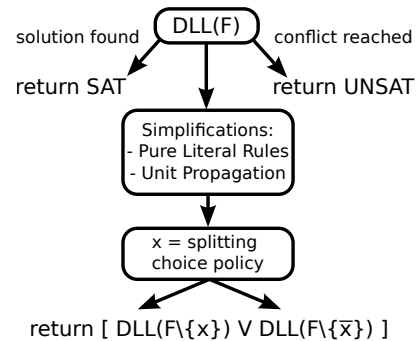


Fig. 1: DLL procedure

2.2 SAT within a parallel context

During the last decade, several works dealt with the parallelization of tree-based searches. To date, two main ways exist for parallel SAT solving : the “divide and Conquer” and the concurrent approaches. In the following section, we propose to enhance the divide and conquer scheme to a new collaborative approach especially designed to shared-memory architecture.

- The *Divide-and-Conquer* (short for DnC) scheme consists in a master-slave model where the master, thanks to a division policy, distributes sub-tasks among the processors. To design a DnC approach, the parallelization of a binary search-tree can be based on different granularities, from a very fine-grained parallelization on a node level to a coarse-grained algorithm on all a sub-tree level. Unless to be able to make an equal-partition of the amount of work into sub-tasks, the main problem with this scheme is the workload-balancing. Thus, computer scientist has to keep it in mind unless it could become a bottleneck for the performances.
- The *Concurrent* scheme consists in distributing several instances of the same task among the processors. Then, each processor must have its own behavior to sequentially solve the problem. Thus, this parallelism aims at selecting the best sequential run. The key-feature is to have several methods to solve the same instance of a problem or to have an algorithm which is able to behave differently according to some parameters. In this latter case, the behavior could be related to

some randomization features. Two types of algorithms are relevant to that characteristic for SAT solving: the incomplete solvers and the CDCL-based algorithms. The two main problems with this scheme are, first to find the right parameters configuration which will reach an efficient enough sequential runtime, and second to be able to generate enough distinct behaviors so that it uses all the cores.

On top of these techniques, a sharing information system may be set up. The shared informations depend on what can be extracted from the search processes. To date, the most popular knowledges produced during SAT solving is the clause learning. Independently of the used scheme, sharing information between the processors aims at helping each of them so that it will speed up its own solving.

2.3 State of the art of parallel SAT solving

Numerous works improving the solving of different class of SAT problems have been proposed and have allowed SAT solvers to be very efficient in processing formulas from which the size and the difficulty increase. Nevertheless, there is to date few parallel solving approaches dedicated to the SAT problem. Moreover, the most of them were designed few years ago and therefore dedicated to the message passing paradigm using a search-space partitioning to assign work to the available processors. This often leads to use a master-slave scheme where the most difficult part consists in workload-balancing. Among the parallel SAT solvers we can remark `PSATO` [24] (in 1996, based on the sequential solver `SATO`) that uses the message-passing paradigm and introduces the notion of *guiding path*. The guiding path is a dynamic object associated to the DLL procedure and represents the partial ordered interpretation of the splitting variables from the root to the current leaf of the search-tree during the backtrack-search process. Thus, it defines disjoint search-spaces respectively assigned to parallel tasks. The fig. 2 represents a sample illustration. Each CPU solves sub-tasks rooted at different dangling nodes of the guiding path. The following other noticeable parallel SAT solvers are based on the guiding path: `//satz` [16], `GridSAT` [8] and `ySAT` [13].

More recently, the interest of the SAT community in speeding up the solving exploiting shared-memory architectures has grown. The actual trend is to propose parallel SAT solvers dedicated to instances from industrial problems. Since determining the best way to set up the solving is not known, the concurrent paradigm is preferred. Studying the relation between the formula and the parameters configuration of the solver is not the topic of this paper but should be essential and useful to speed-up future SAT solvers. Among multi-threaded SAT solvers from the literature, `ManySat` and `Plingeling` are the best known. `ManySat` [14] is a multi-threaded concurrent implementation of the sequential solver `MiniSat v2.02` [22] where is grafted some extensions

of conflict-analysis. `Plingeling` [4] is a multi-threaded version of `Lingeling`. This port-folio approach shares only the unit clauses generated during the CDCL processing.

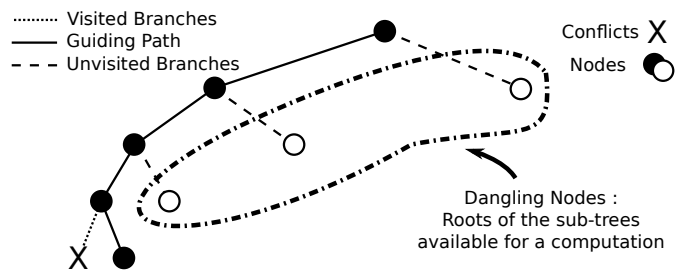


Fig. 2: Guiding Path Sample

These recent solvers must have as many different behaviors as available cores and the growth of cores should continue in the future. In that context, and despite the trend in concurrent scheme, we propose to enhance the DnC policy to a more collaborative approach designed for the shared-memory architecture particularly by removing the master/slave feature.

3. A collaborative multi-threaded SAT solver

3.1 The needs

The objective is to design an efficient (in term of parallelism) SAT solver for the multi-cores architectures. It is essential to keep in mind the number of cores will increase over the years. The idea is to preserve a speed-up factor according to the number of cores. The concurrent approach can't insure that because of the difficulty to design or choose a high enough number of distinct behaviors. Moreover, a lot of improvements have been proposed and are grafted to existing sequential SAT solvers. Thanks to these assessments, our choice is to deal with collaborative scheme.

It is necessary to preserve the contributions propose by the SAT community. In that way, the solution is to have a very fine grained application. Instead of creating a parallel version of each local treatment, each splitting choice policy, *etc.* which would be a very fine grained solution but very hard to design and not reusable in the future, we choose to parallelize the binary search-tree at a node level or even smaller as the sequence of local treatments. The very fine grained solver able to follow the current industry choices in term of processors must divide the job by distributing the nodes of the search-tree. The main problem is the number of tasks: several thousands nodes per second could usually be explored by a SAT solver. This leads to a massive concurrent access to the job sharing structure. Moreover having a fine grained algorithm on the search-tree gives the ability to adapt the granularity as needed. The algorithm should be highly flexible, adaptable and robust. The main idea behind this

flexibility is to allow the community to easily parallelize a sequential SAT solver since it is the most natural way to preserve the efficiency of the current SAT solvers.

3.2 The guiding tree approach

Parallelizations of algorithms such as DLL were previously made using the guiding path structure. The guiding path is the path of a particular processor. The available tasks for other processors are the dangling sub-trees on this path. Each path is given to a processor, it indicates the sub-tree on which a processor should work thanks to the path from the root of the search-tree to the root of the sub-tree. This structure cannot support a too fine granularity since the number of generated tasks is huge and each path as to be explicitly given to a processor. The threads dedicated to this work will be overloaded. Then, the most important point is the number of available tasks extracted from this structure. Currently, the maximal depth of a binary search-tree is linear according to the number of variables but the size of the search-space is exponential. That is to say the tasks are very unbalanced and cannot be distributed on a large number of processors. More precisely even on big formulas, the guiding path can not offer more than 35 tasks. Hence, the work distribution is quite unfair.

To replace the guiding path as a tasks distributing and workload-balancing tool, we propose a new structure. With a very fine grained application, an explicit workload-balancing is not needed because the tasks are all balanced and non divisible. The time needed by each task is very short: few milliseconds. Thus, we introduce the guiding tree approach which can be seen as a parallel-built tree that is memory shared. The guiding tree allows to distribute tasks as small as a local treatment at a node. While keeping a sequential computation for each task, which is essential to preserve the SAT engineering since more than 20 years, becomes possible to parallelize the computation.

The main idea behind the guiding tree is to design within a parallel context search-tree areas that are not explored. Thus, designing this object is both solving the problem and balancing the workload. Each thread starts its processing by looking for an open node of the guiding tree. It then does the associated job, puts the result down in the tree and opens a new node and so on. The flexibility of the guiding tree is obtained by the fact the thread chooses the size of the job thanks to its configuration and/or according to the informations in the chosen node.

With a huge amount of small tasks that generate a huge number of input/output, using a specific structure to distribute the work is a bottleneck. The size of the guiding tree structure increases with the number of available tasks: each leaf is an available work. In that context, the idle threads can initiate a work on its own. Thus, several threads can initiate a work in distinct search-tree areas at the same time. Larger

is the problem, larger is the guiding tree and lower conflicts are encountered.

The flexibility of the guiding tree allows to implement hybridization with complete or incomplete searches, mixed architectures, fine or coarse grained parallelism, threads can have specific functions, ... The most important point is that flexibility is achieved while maintaining a good parallel efficiency till the tasks are small enough. Indeed, a coarse grained parallelism is possible using the guiding tree but the workload-balancing is no longer assured since it relies on the principle of extremely small tasks.

4. MTSS

We designed a new multi-threaded SAT solver that implements the guiding tree. The available tasks in MTSS are:

- The propagation of the truth value of a variable in the formula and the splitting choice policy.
- The look-ahead, a classic local treatment in SAT, which consists in checking if at least one of the truth value related to a variable reaches a conflict. Moreover, the look-ahead leads to open one node in the guiding tree.
- The computation of an entire sub-tree by an external SAT solver.

MTSS is a complete SAT solver. To design a complete algorithm using the guiding tree, one specific thread is needed so that it draws a limit between the computed sub-tree and the remaining search-tree. This thread is called the *rich thread* (algorithm 1), the other threads are the *poor threads* (algorithm 2). The rich thread corresponds to a kind of the classical DLL procedure. If the rich thread finds informations computed by one or more poor threads, it replaces its current context with the poor computed one (see *Context Swap* label). If an information is under construction by a poor. The rich does not wait for the end and mutates into a poor. It previously indicates that it has been changed its role at the current node. At this moment, there is no rich (see *Role Swap* label). The poor threads compute tasks on the guiding tree without following a rule in their moves. Figure 3 represents a guiding tree developed by MTSS.

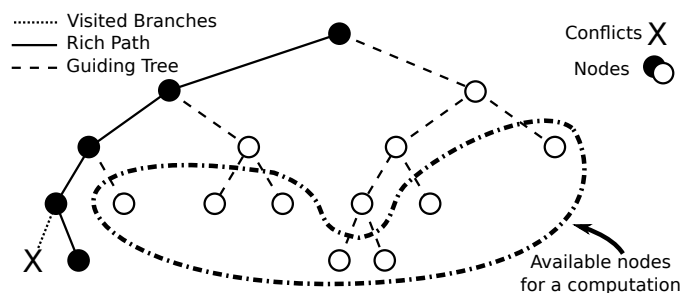


Fig. 3: Guiding Tree Sample in MTSS

For the use of an external solver, MTSS deploys a guiding tree from the root to the calls of the external solver. The

Algorithm 1 MTSS RICH THREAD procedure

Require: \mathcal{F} , a propositional formula
 RICHTHREAD(\mathcal{F})
if \mathcal{F} contains one monotonic literal l **then**
 Return RICHTHREAD($\mathcal{F} \setminus l$) (**Monotonic Literal**)
else if \mathcal{F} contains one unit clause containing l **then**
 Return RICHTHREAD($\mathcal{F} \setminus l$) (**Unit Propagation**)
else if \mathcal{F} contains at least one empty clause **then**
 Return FALSE (**Backtrack**)
else if \mathcal{F} is empty **then**
 Return TRUE (**Solution**)
else
 $v \leftarrow$ one unassigned variable of \mathcal{F} (**Split**)
 if RICHTHREAD($\mathcal{F} \setminus v$) = TRUE **then**
 Return TRUE
 else if At least one Poor Thread has finished its local
 calculus on the current node **then**
 Replace current computing context by the Poor Thread's one (**Context Swap**)
 else if Current node is in the course of a Poor Thread
 calculus **then**
 Indicate a Role Swap on the current node
 Become a POOR THREAD (**Role Swap**)
 else
 Return RICHTHREAD($\mathcal{F} \setminus \bar{v}$) (**DLL Backtrack**)
 end if
end if

external SAT solver must solve the entire sub-tree for each mutation. That means MTSS can solve a formula from a very fine grained parallelism to a coarse grained parallelism.

Even if MTSS is originally designed for the randomly generated formulas, it is able to parallelize different classes of complete solvers. Hence, each complete sequential SAT solver thanks to an hybridization with MTSS is able to take advantage of multi-core architecture. To increase performances of the CDCL-based algorithms, it is needed to share the informations extracted during the computation. Thus, MTSS includes a technology receiving and sharing the learnt clauses by the external SAT solvers. In order to graft this ability to a sequential SAT solver, a library is proposed on a dedicated website². Let n the number of executions of the external SAT solver at the same time. When MTSS receives the learnt clauses sent by the external SAT solver $j \in \{1, \dots, n\}$, it adds them in a shared database and then sends the learnt clauses received by the other executions of the external solver $i \in \{1, \dots, n\} / i \neq j$ since the last sent to j . The main result about this parallelization of the sequential SAT solver is the super-linear speed-ups observed in some cases. Actually the search-space is sometimes reduced thanks to the shared learnt clauses. More details

²www.parallel-sat.net

Algorithm 2 MTSS POOR THREAD procedure

Require: \mathcal{F} , a propositional formula
Require: T, a task
 POORTHREAD(\mathcal{F} , T)
 $n \leftarrow$ Root of \mathcal{F} -search-tree
while \mathcal{F} has no solution **do**
 if T can be applied on n **then**
 Apply T on n
 if The Rich Task Indicates a Role Swap on n **then**
 Become the RICH THREAD (**Role Swap**)
 end if
 end if
 if n is the node in computation by the RICH THREAD
 then
 $n \leftarrow$ Root of \mathcal{F} -search-tree
 else
 Extends the *guiding sub-tree* rooted in n (**Guiding Tree Extension**)
 $n \leftarrow$ next node in the *guiding path*
 end if
end while

about this topic and other aspects of MTSS are given in [23].

5. Experiments

5.1 Protocol

MTSS is written in C language with PTHREADS functions and was compiled by ICC 11.1 (an OPENMP version obtains the same performances). The cluster of SMPs used for benchmarks is CLOVIS³ from the University of Reims Champagne Ardenne. 472 cores are dedicated to computation. Among others nodes, it contains 36 nodes of 12 cores (2 * Westmere-EP) and 24 Gb of memory, we mainly used these nodes. Two series of runs were launched to estimate the quality of the snap-parallelization of sequential solvers. The first one is about the randomly generated formulas parallelizing two of the best solvers for this type of formulas. The next one is about the industrial formulas parallelizing the most famous sequential SAT solver for this task. We compare our parallelization to the current best parallel SAT solvers using the concurrent approach and tuned to solve the industrial formulas.

5.2 Randomly generated formulas

The randomly generated formulas solved are 3-SAT with 500 variables generated at the pick of difficulty (2125 clauses). Both SAT and UNSAT formulas (a dozen by type) were solved but we give in figure 4 only the mean computation time for the UNSAT formulas. Indeed, the UNSAT

³www.romeo2.fr

formulas have a better interest from a comparative perspective since the solvers have to develop the entire search-tree. The lines from 1 to 12 cores represent the sequential solvers we used: `March` [15] and `Kcnfs` [11]. They are both excellent and competitive complete SAT solvers for randomly generated formulas. The curves in that figure represent the mean computation time for MTSS and `March` or `Kcnfs` parallelized by MTSS. Each solver was run once on a formula and the number of cores tested were: 1, 2, 4, 8 and 12 (so many threads have been launched as many cores were used, we used one rich thread and the remaining ones were poor threads). Speed-up with 12 cores for MTSS is 11,5 (efficiency of 95.85%), for `March` parallelized by MTSS, it is 7,53 (eff. 62.78%) and for `Kcnfs` parallelized by MTSS, it is 6,78 (eff. 56.53%). It is interesting to notice these programs were not modified and the robustness is excellent: between two runs, the difference of computation time is very light. This robustness is true for runs on randomly generated formulas, indeed MTSS alone is extremely robust.

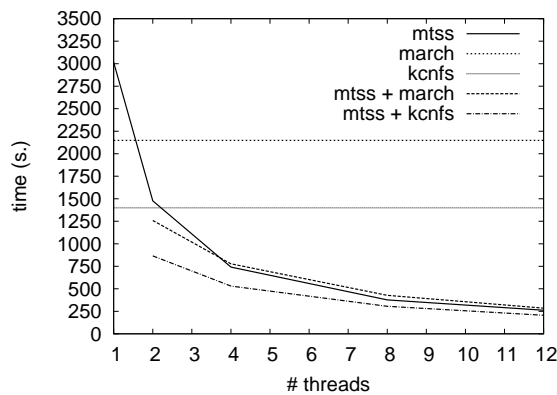


Fig. 4: Mean Computation Time for UNSAT 3-SAT formulas (500 variables)

5.3 Industrial formulas

163 formulas from the SAT competition were used for this benchmark. Among them, 59 are SAT, 57 are UNSAT and 47 formulas were not proved SAT nor UNSAT during this test. We launched the sequential SAT solver `Minisat` on one core and we parallelized it by MTSS on 12 cores. We also compare this parallelization to `Manysat` and `Plingeling`. `Manysat` is a concurrent SAT solver version of `Minisat` with a sharing system policy while `Plingeling` has the same principle but for the sequential SAT solver `Lingeling` with an extremely light sharing policy. These three solvers are all designed to solve the industrial formulas whereas MTSS is not. MTSS is potentially able to parallelize every solvers. Moreover, in the case of `Minisat`, it shares the learnt clauses between its different runs. The modification of `Minisat` is quite simple and we propose a library to set it up in any solver. A representative sample of these runs is

given in table 1. Each line represents a set of formulas or one formula. The number of solved formulas and the mean solving time is given for each set and for each solver. A timeout of 1200 seconds was set up for this test in spite to tie in with the SAT competition test conditions. The time is given in seconds and is the best runtime among 10 runs.

On formulas called *gus-md5-*.cnf* and *AproVE09-*.cnf* (but 20), you can remark that the trees developed by `Minisat` are smaller than those developed by `Plingeling` even on 12 cores. At least one of the trees developed by `Manysat` has a size in about the `Minisat` ones. In these cases, MTSS has the best computation times. That means MTSS is efficient to partition and parallelize the `Minisat`'s trees. This situation happens on SAT or UNSAT formulas. It is noticeable the cryptographic benchmarks are well managed by MTSS. Even if it seems trivial MTSS can divide the size of the tree developed by MTSS. It was not so simple because MTSS is designed for randomly generated formulas. It doesn't restart its tree and hence the guiding tree part is never erased during the process, whereas all the sub-trees computed by `Minisat` are periodically erased by restarts. In the case of the formula called *AProVE09-20.cnf*, `Minisat` develops a huge tree and MTSS is still able to develop a smaller tree. At least one of the trees developed by `Manysat` and `Plingeling` for each solver is smaller any more. On the formulas in the *q_query-*.cnf* family MTSS is quite good but the original trees from `Minisat` are very large. In this case, the concurrent approach succeeds to develop smaller trees. About the formulas from the families *Set 1* and *post-*.cnf*, MTSS solves more formulas than `Minisat` but the computation time is not as good as the one of the other families. In *Set 2*, `Minisat` develops so huge trees than even parallelized by MTSS, the tree is not explored entirely by 1200 seconds. However the concurrent approaches success in this task. The last interesting sample is the formula *UR-10-5pl.cnf* where the MTSS's tree is larger than the `Minisat`'s one. That shows sometimes the guiding tree part, only managed by MTSS, forces `Minisat` into sub-trees bigger than the one it could develop alone. That can be explained with the MTSS splitting policy that is not adapted with the case of industrial solving. The collaborative approach makes sense for the industrial formulas when the sequential tree developed is quite small. Finally, the lack of robustness of the CDCL-based solvers is illustrated when comparing `Manysat` to `Plingeling`. Indeed, `Plingeling` shares a weak amount of learnt clauses, compared to `Manysat` but remains the best strategy.

Moreover, in [23], we gave some results about super-linear speed-ups about SAT formulas and UNSAT formulas. Some explanations and more details about the parallelization and the sharing system are given in this paper.

Table 1: Computation Times on Industrial Formulas

family/formula	value	# form.	1 core		12 cores					
			Minisat		MTSS + Minisat		Manysat		Plingeling	
			# solved	time	# solved	time	# solved	time	# solved	time
AproVE09-*.cnf (but 20)	SAT	17	17	0.99	17	0.27	17	1.21	17	3.20
AProVE09-20.cnf	SAT	1	1	925.85	1	68.68	1	25.72	1	4.49
Set 1	SAT	4	1	723.30	4	668.01	4	218.66	4	67.95
Set 2	SAT	4	0	-	0	-	4	575.76	4	181.53
UR-10-5p1.cnf	SAT	1	1	6.93	1	10.85	1	5.46	1	4.62
gss-*-s100.cnf	SAT	11	6	332.86	8	44.81	9	164.43	11	89.58
gus-md5-*.cnf	UNSAT	6	5	373.19	6	135.64	5	354.69	4	649.33
q_query_*.cnf	UNSAT	11	10	597.92	11	423.50	11	151.54	11	109.70
post-*.cnf	UNSAT	7	2	302.20	3	424.13	5	241.96	7	146.55

6. Conclusions and perspectives

The guiding tree approach extends the DnC principle to permit a strong collaborative work between cores developing a parallel search-tree. It is based on a very fine grained parallelism. It is flexible during the computation in spite of integrating all the very efficient techniques embedded in the current sequential SAT solvers, and offering an implicit workload-balancing. Thanks to this flexibility, the guiding tree approach is able to parallelize a sequential SAT solver. The solver designed on this basis is called MTSS and is originally designed to solve the randomly generated formulas. Benchmarks on these formulas show good results as the efficiency of MTSS is more than 95% on 12 cores. MTSS is also efficient parallelizing solvers dedicated to the randomly generated formulas. MTSS can divide the search-space of a solver dedicated to solve the industrial formulas and shares the learnt clauses between the runs. Finally these snap-parallelizations are quite simple to set up.

The guiding tree approach is rich of its possibilities. The flexibility offers a lot of perspectives. Among the main ones, it could be interesting to imagine new ways to extract informations from the solving. The aim is to share them among the threads to reduce the size of the guiding tree. Since the CDCL-based solvers are not robust, MTSS could be improved by parallelizing a solver with different presets or finding a way to determine the right configuration of a solver in spite of having a small tree to divide. Finally, with a dedicated version of MTSS for the industrial formulas, we could reach very good performances on these formulas. The randomly generated formulas are well managed by MTSS even by parallelizing an external sequential SAT solver.

References

- [1] Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. *SAT*, 2003.
- [2] A. Bhalla, I. Lynce, J. T. Sousa, and J. Marques-Silva. Heuristic-based backtracking relaxation for propositional satisfiability. *J. Autom. Reason.*, 35(1-3):3–24, 2005.
- [3] A. Biere. Adaptive restart strategies for conflict driven sat solvers. *SAT 2008*, pages 28–33, 2008.
- [4] A. Biere. Lingeling, plingeling, picosat and precosat. Technical report, Solver Description, SAT-Race 2010, 2010.
- [5] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
- [6] Armin Biere, Keijo Heljanko, Tommi Junttila, Timo Latvala, and Viktor Schuppan. Linear encodings of bounded LTL model checking. *Logical Methods in Computer Science*, 2, 2006.
- [7] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation: An algorithm for satisfiability. *Random Struct. Algorithms*, 27(2):201–226, 2005.
- [8] W. Chrabakh and R. Wolski. Gridsat: Design and implementation of a computational grid application. *J. Grid Comput.*, 4(2):177–193, 2006.
- [9] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, 1971.
- [10] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [11] Gilles Dequen and Olivier Dubois. An efficient approach to solving random sat problems. *J. Autom. Reasoning*, 37(4):261–276, 2006.
- [12] Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-sat formulae, 2001.
- [13] Y. Feldman, N. Dershowitz, and Z. Hanna. Parallel multithreaded satisfiability solver: Design and implementation, 2004.
- [14] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2009.
- [15] Marijn Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *JSAT*, 2(1-4):47–59, 2006.
- [16] B. Jurkowiak, C. M. Li, and G. Utard. Parallelizing Satz Using Dynamic Workload Balancing. In *SAT 2001*, 2001.
- [17] H. Kautz and B. Selman. Pushing the Envelope: Planning, Propositional Logic and Stochastic Search. In *Proc. of 30th national AI and 8th IAAI*, 1996.
- [18] I. Lynce and J. Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. *Tools with Artificial Intelligence, IEEE International Conference on*, 2003.
- [19] D. Mitchell, B. Selman, and H. J. Levesque. Hard and easy distribution of SAT problems. In *Proc. 10th Nat. Conf. on Artificial Intelligence*, pages 459–465. AAAI, 1992.
- [20] N. R. Potlappally, A. Raghunathan, S. Ravi, N. K. Jha, and R. B. Lee. Aiding side-channel attacks on cryptographic software with satisfiability-based analysis. *IEEE Trans. VLSI Syst.*, 2007.
- [21] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Proceedings of the Second DIMACS Challenge on Cliques, Coloring, and Satisfiability*, 1996.
- [22] N. Sörensson and N. Eén. Minisat 2.1 and minisat++ 1.0. Technical report, SAT-Race 2008: Solver Descriptions, 2008.
- [23] P. Vander-Swalmen, G. Dequen, and M. Krajecki. Toward easy parallel sat solving. In *21st ICTAI*, 2009.
- [24] H. Zhang, M. P. Bonacina, and J. Hsiang. Psato: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 1996.
- [25] L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of ICCAD*, 2001.

On Using a Graphics Processing Unit to Solve The Closest Substring Problem

Jon Calhoun^{1,2}, Josh Graham¹, and Hai Jiang¹

¹Dept. of Computer Science, Arkansas State University, Jonesboro, AR, US

²Dept. of Mathematics and Statistics, Arkansas State University, Jonesboro, AR, US

Abstract—Finding a string that is close to another is a common dilemma in computational molecular biology and many other fields. The problem comes in two varieties; closest string (CSP), and closest substring (CSSP). The computational complexity increases exponentially as the data-set size increases. We make use of a massively parallel algorithm and the parallel nature of a graphics processing unit (GPU) in order to flatten the data-set size verses time curve and enable more applications to calculate results in reasonable time. In this paper we focus on CSSP and show that GPU devices can be used to reduce the time needed to find the closest substring. We examine an exact algorithm and extract independent parts in order to form a massively parallel interpretation of the sequential algorithm. We contribute a fast, exact, algorithm that can solve the CSSP much faster than sequential versions.

Keywords: closest substring problem, GPU, CUDA

1. Introduction

Closest substring problem (CSSP) is a common open problem in many applications. The closest substring problem was introduced in [3] and is a key theoretical open problem in applications such as antisense drug design, creating diagnostic probes, and creating universal PCR primers [1]. Many applications would benefit from a faster algorithm to find the closest substring. Some applications can accept approximations, but others need an exact result. We focus on using GPU devices to speed up an exact CSSP algorithm.

The CSSP is an NP-hard problem [3], and can be defined formally.

- Let Σ be a fixed finite alphabet.
- Let s and s' be finite strings over Σ .
- Let $d(s, s')$ denote the Hamming Distance between s and s' .
- Given a set $S = \{s_1, s_2, \dots, s_n\}$ of strings each of length m . Find a center string c of length L minimizing d such that for each s_i in S there is a length L substring t_i of s_i with $d(c, t_i) \leq d$.

Solving the closest substring problem is moderately difficult, but solving it efficiently has proven very difficult. There is a tremendous amount of computation to be done and the process is compounded by not only being required to find a solution within the tolerance but to find the best solution. Solving the same problem in parallel can be taxing

on one's logical skills. Additionally, solving the problem by hand is extremely time consuming for all but trivially sized data-sets. This paper shows that for applications needing an exact solution to the CSSP in a small amount of time can use GPUs to solve the problem, and achieve significant speedups.

This paper makes the following contributions:

- An efficient exact algorithm for computing the closest substring on GPU. We extract parallel sections of the computation in order to have it run efficiently.
- Logical configuration for launching the algorithm as a CUDA kernel.
- Experiment results that demonstrate the efficiency of the algorithm.

In Section 2, *Background*, we will explain the various technologies used. Following, Section 3.1, is a description of how the CSSP can be solved on a CPU using sequential code. After the CPU algorithm is clearly spelled out we turn to detail the inner workings of three parallel algorithms and their benefits and pitfalls in Section 3.2 - 3.3. In Section 4, *Experimental results*, we compare our GPU algorithms to the sequential one and discuss the performance increase. Section 5, *Related work*, discusses works that apply the GPU to the CSSP and related problems. A brief summary of what was learned from interpreting Section 4 can be found in our *Conclusion*, Section 6. Finally, in Section 7, *Future work*, we discuss optimizations that could be implemented that may increase performance.

2. Background

2.1 CSSP

The CSSP is a common problem in many areas. Particularly in computational biology, the CSP and CSSP have found numerous practical applications such as identifying regulatory motifs and approximate gene clusters, and in degenerate primer design [7]. The CSSP problem is much more elusive than the Closest String problem [6]. Many people have studied approximation algorithms for CSSP and there has even been work done on an evolutionary algorithm [8]. Here, we study an exact algorithm.

Many problems in molecular biology involve finding similar regions common to each sequence in a given set of DNA, RNA, or protein sequences. These problems find applications in locating binding sites and finding conserved regions in

unaligned sequences, genetic drug target identification, designing genetic probes, universal PCR primer design, and, outside computational biology, in coding theory [6]. Such problems may be considered to be various generalizations of the common substring problem, allowing errors [6].

2.2 GPU

In 1999, NVIDIA created and marketed the worlds first graphics processing unit (GPU). Since then, there have been yearly breakthroughs in GPU technology. With the need to be able to make thousands of intense calculations per second for graphic applications. The architecture of a GPU is composed of thousands of processing units. In order for a algorithm to run efficiently on a GPU, the algorithm must be massively parallel. It is no surprise then, that the hardware and capabilities of GPUs has improved dramatically since their inception.

Programmers have been aware of the performance gain that could be achieved if a parallel portion of a program was executed on the GPU, but it was not until NVIDIA released there Compute Unified Device Architecture (CUDA) language that the job of programming GPUs became more intuitive. Before then to access the computational resources, a programmer had to cast his or her problem into native graphics operations so the computation could be launched through OpenGL or DirectX API calls [5].

2.3 CUDA

CUDA is NVIDIA's parallel computing language. It enables dramatic increases in computing performance by harnessing the power of the GPU. When NVIDIA introduced the GeForce 8800 GTX in November 2006 the CUDA architecture debuted. This architecture included several new components designed strictly for GPU computing and aimed to alleviate many of the limitations that prevented previous graphics processors from being legitimately useful for general-purpose computation [4].

CUDA is the most widely adopted programming platform for GPU development. CUDA applications running on NVIDIA graphics processors enjoy superior performance per dollar and performance per watt than implementations built exclusively on traditional central processing technologies [4]. In the CUDA programming model, GPUs which are called devices, execute highly parallel portions of an application, called kernels which are made up of many threads working cooperatively. CUDA permits the programmer to use different memory spaces explicitly. Examples of these different memory spaces include: global, shared, constant, and texture memory. Each space has its own performance advantages and penalties.

NVIDIA introduced the Fermi architecture recently. Fermi brings in many new capabilities. In this paper we make use of the increased maximum number of threads and blocks to perform more cooperative computations. We also benefit from the faster atomic actions and large memory present in Fermi graphic cards.

3. Algorithms

In order to ease the following discussions we define some terms.

- Let a *window* be any substring of length L from a given string.
- Let a *pitch* be special window from the first string that other notes are compared against.
- Let a *note* be the window from a string that is closest to the pitch.
- Let a *chord* be a set of notes, one per string, closest to a pitch.
- Let the *chord distance* be the sum of all note distances in a chord from the pitch.
- Let the *root* be the average of a chord.

3.1 CPU algorithm (CPU)

The strategy for finding the closest substring on the CPU consist of taking each pitch from the first string, comparing it against all windows, in all other strings and finding the closest window in each string. This window is then deemed a note and is part of the chord based on the pitch taken from the first string.

Given Figure 1, we want to find that something very close to "gcc" occurs in every string.

```
agccatt
ggaagcc
aagtctg
```

Fig. 1
EXAMPLE INPUT.

We do so by fixing a pitch in string 1 and comparing all others against it. Then we move to pitch 2 and so on. As demonstrated in Figure 2.

```
agccatt  →  agccatt
ggaagcc  →  ggaagcc
aagtctg  →  aagtctg  →  ...
```

Fig. 2
ILLUSTRATION OF EXECUTION.

From the search we get the best chord, shown in Figure 3.

```
gcc
gcc
gtc
```

Fig. 3
BEST CHORD FROM DATA SET.

In Figure 4, we average the chord to determine the closest substring. In this example, the second character of each of the notes is *c*, *c*, and *t* respectively. In the averaging process *c* will be chosen because it occurs more often than any other character.

```

gcc
gcc
φ gtc
gcc

```

Fig. 4

EXAMPLE OF AVERAGING A CHORD TO FIND A ROOT NOTE.

To solve the problem more formally,

- Let $S = \{s_1, s_2, \dots, s_n\}$ be the set of all strings.
- Let Σ be a fixed finite alphabet.
- Let $S_i = \{s_{i_1}, s_{i_2}, \dots, s_{i_n}\}$ be the set of all windows in $S_i \mid \forall s_{i_{j_k}} \in \Sigma$.
- Let d be the maximum distance.
- Let L be the length of the substrings.
- Let q be the number of windows in a single string.
- Let $P = \{p_1, p_2, \dots, p_q\}$ be all the pitches from s_1 .
- Let $T = \{t_1, t_2, \dots, t_q\}$ be the set of all chords.
- Let $T_i = \{t_{i_1}, t_{i_2}, \dots, t_{i_q}\}$ be the set of notes composing a chord.
- Let $B = \{b_1, b_2, \dots, b_q\}$ be the set of all chord distances.
- Let $T_{i_j} = \{t_{i_{j_1}}, t_{i_{j_2}}, \dots, t_{i_{j_L}}\}$ be the set of characters in a single note.
- Let $C = \{c_1, c_2, \dots, c_L\}$ be the characters composing closest substring.
- Let $\varphi(op_1, op_2, \dots, op_n)$ denote picking the most common element from a collection.
- Let $+$ denote character concatenation.
- Let $d(s_1, s_2)$ denote the hamming distance between s_1 and s_2 .
- Let $k \in \mathbb{N}, [1, q]$

$$if \exists t_i \in T \mid (\forall t_{i_j} \in t_i \exists s_{i_j} \in S_i \mid d(p_k, s_{i_j}) \leq d \wedge d(p_k, s_{i_j}) \leq d(p_k, S_i)) \wedge (b_i \leq B)$$

$$then c = \varphi(t_{i_{1_1}}, t_{i_{2_1}}, \dots, t_{i_{q_1}}) + \varphi(t_{i_{1_2}}, t_{i_{2_2}}, \dots, t_{i_{q_2}}) + \dots + \varphi(t_{i_{1_L}}, t_{i_{2_L}}, \dots, t_{i_{q_L}})$$

Applying big-O analysis to the algorithm yields, $O((k-1)n^2)$ where k is the number of lines, and n is the number of windows in a string. A sequential pseudocode algorithm for the CPU can be seen if Figure 5.

```

for each pitch
  for each string
    for each window
      distance = hammingDistance(pitch, window)
      if distance is minimum
        minDistance = distance
    if note found
      numNotesFound++
      chordDistance += minDistance
  if chordDistance is minimum
    bestChord = currentChord

root = calculateRootNote(bestChord)

```

Fig. 5

CPU ALGORITHM PSEUDOCODE.

3.2 Purely parallel GPU algorithm (PP-GPU)

An observation can be made about the sequential algorithm, each window's hamming distance to a certain pitch in the data set is completely independent of all the others. This observation implies that the parallel nature of the CUDA language can be exploited while calculating the hamming distance of all the windows with respect to pitches. An illustration of this is shown in Figure 6. If looking strictly at parallel computation even the process of computing the hamming distance itself can be incorporated into our CUDA algorithm (PP-GPU).

```

agccatt  agccatt
ggaagcc + ggaagcc ...
aagtctg  aagtctg

```

Fig. 6

PP-GPU ALGORITHM DESIGN.

3.2.1 PP-GPU

Initially our strategy was to exploit the parallel nature of CUDA by calculating each window's hamming distance to every pitch concurrently, while at the same time calculating the hamming distance in parallel. The kernel grid was aligned to perform the error calculation in one kernel. The X direction of the grid being the number of windows per row, $x = q$, signifying which window in the row we are, and the Y direction being number of rows, not including the row of pitches, multiplied by the number of windows per row, $y = numRows * q$. The Y direction was aligned in such a way so that the blocks Y coordinate modulo number of rows, not including the row of pitches, provides the row in the data set, the window that this block simulates, $by = blockIdx.y \% numRows$ and the blocks Y coordinate divided by number of rows, not including the row of pitches, yields the pitch we are to perform the hamming distance on $i = blockIdx.y / numRows$. If the block index was $(0, 4)$ in the 3×7 data set listed above, in Figure 6, then the block corresponds to the following where the pitch is on the first row and the window is on the second as shown by Figure 7.

```

agccatt
ggaagcc
aagtctg

```

Fig. 7

BLOCK INDEXING SAMPLE.

Threads in each block calculate the hamming distance in parallel. A pseudo code version of the algorithm shown in Figure 8.

```

concurrently for all windows:
  tx = threadIdx.x
  bx = blockIdx.x
  by = blockIdx.y % numRows
  i = blockIdx.y / numRows
  result = window[tx] != pitch[tx]
  distance = combine "result" from all threads in the block

chordDistance = Summation of distances of minimal distance to the pitch
locate minimal chordDistance and calculate root

```

Fig. 8

PP-GPU ALGORITHM PSEUDOCODE.

Although this idea provides the greatest amount of parallelism, due to hardware limitations this algorithm proved to be the slowest of the GPU based algorithms. The hardware limitation was with either a large data set and/or a small window size the number of blocks to be placed in the grid out grew the maximum grid limit imposed by CUDA. This required the kernel to be launched several times. With each kernel invocation there is time wasted making the kernel call and with a data set of 512 x 512 characters and window sizes of 15 characters the grid in the Y direction needed 254,976 blocks. However CUDA only supports 65,535 blocks in the Y direction on the grid [2], thus resulting in the kernel needing to be called 4 times, and that number increases to 16 with a 1024 x 1024 data set with the same sized windows.

3.2.2 PP-GPU*

Discovering the hardware limitation gain we attempted to remove multiple kernel launches in an attempt to increase performance by shrinking the number of blocks in the Y direction with the use of a for loop that iterates over the windows in a row. This design came to be known as (PP-GPU*). Incorporating this idea into the above algorithm design we can lay out the grid as to align the Y direction with the rows of the data set while the X is aligned on pitches we are to find a hamming distance to. Each block is still calculating the hamming distance in seemingly parallel. A pseudo code algorithm is shown in Figure 9.

```

concurrently for each pitch:
  for each window in my row
    calculate error for my element of the window to the pitch
    distance = summation of local errors
    if distance < minDistance
      store distance and the windows index
    if minDistance > maxError
      throwout this pitch
  chordDistance = summation of minDistances for the pitch

locate minimal chordDistance and calculate root

```

Fig. 9

PP-GPU* ALGORITHM PSEUDOCODE.

PP-GPU attempted to perform as many calculations as possible in parallel after experimentation was performed in order to optimize the kernel. An optimized version PP-GPU* did remove the hardware limitations and increased the speedup shown in *Experimental Results*, but more performance was possible. The parallel calculation of the hamming distance was discarded and the layout and functions of the grid, block structure was redesigned.

3.3 Streamlined GPU algorithm (S-GPU)

Upon seeing that hardware limitations will be hit if we take advantage of every bit of parallelism present in the problem, we re-engineered and streamlined the algorithm. The resulting algorithm takes the most efficient parallel ideas and discards those that only caused increased overhead when implemented on current hardware.

The main idea behind the streamlined algorithm is that in order to find the closest substring, all chords must be calculated and compared in order to determine which one has the smallest chord distance. Each chord finding operation is independent of each other. Looking further, within each chord finding operation, each note finding operation is independent. These facts point us toward a parallel algorithm in which all chords can be found simultaneously, and within each chord finding operation, all notes can be found concurrently.

The aforementioned description leads to an implementation of an algorithm that essentially eliminates the outer two *for* loops of the CPU algorithm, by doing them concurrently. We chose to have each CUDA block calculate the best chord associated with a single pitch, each CUDA thread of the block will search one string for the closest note concurrently. This allows us to take advantage of the CUDA programming model that allows threads to cooperate. In CUDA, when a kernel is called the caller must specify the number of blocks that will execute the kernel and the number of threads that each block should contain. Our kernel uses the number of windows possible in a string to be the number of blocks, specifically $numBlocks = q$. The number of threads is directly proportional to the number of strings, specifically $numStrings = numThreads$. The chosen configuration allows for notes of a chord to share their distances so that a chord distance can be calculated, CUDA threads allow this

kind of cooperation. This configuration lends itself well to the problem and increases parallelism without adding additional memory requirements. Figure 10 shows a simplified version of the algorithm in pseudocode.

```

each string with each pitch concurrently:
  for all windows in a string
    distance = hammingDistance(pitch, window)
    if distance is minimum
      minDistance = distance
      noteFound = 1
  add(numNotesFound, noteFound)
sync
if numNotesFound == numRows
  add(chordDistance, minDistance)
else
  invalidateChord()
    
```

Fig. 10
S-GPU ALGORITHM PSEUDOCODE.

This, Streamlined GPU algorithm results in less memory use on the device. Simply put, the memory use went from using n^2 amount of memory to store intermediate results with PP-GPU, to using only n memory for intermediate results on both PP-GPU* and S-GPU. Where n is the size of the data-set. This decrease is tremendous when considering inputs for n are typically large. It should also be pointed out that S-GPU cuts the size of the results generated from the kernel from n^2 to n . This is good news considering that the transfer from CPU to GPU memory has historically been a bottleneck. The streamlined algorithm also allows for one kernel to do all necessary calculations, rather than multiple kernels which was required in the PP-GPU algorithm. These improvements result in a faster algorithm.

4. Experimental results

4.1 Test hardware

All experiments were performed on the following system:

- CPU: 2x Intel Xeon X5660 @ 2.80GHz
 - 6 core 12 threads per CPU
- Memory: 24 GB
- GPU: Tesla C2070 @ 1.15 GHz
 - Driver version: 260.19.26

4.2 Experiments

4.2.1 Experiment 1

Compared to another exact version (running sequentially), we achieved a 18x speedup for an input file of size 1024 x 1024 with a length of 3 as shown in Figure 11.

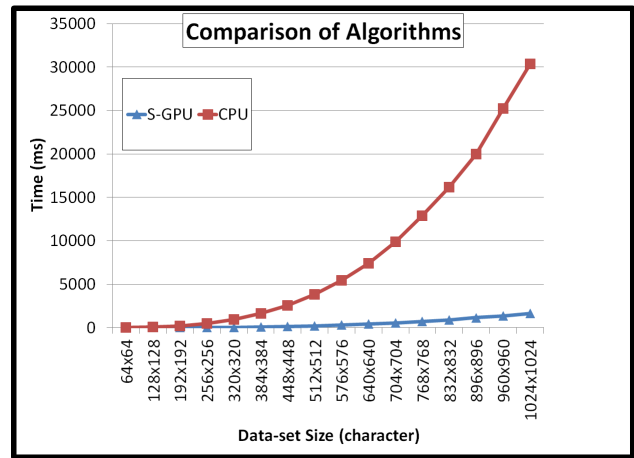


Fig. 11
COMPARISON OF ALGORITHMS CPU AND S-GPU:
 $L = 3, d = 3$

By successfully flattening the time vs. data-set size function it is possible for some applications to get timely results whereas before results would have been prohibitively expensive. The accuracy of the algorithm is unchanged. It still can deduce the closest substring with precise accuracy.

4.2.2 Experiment 2

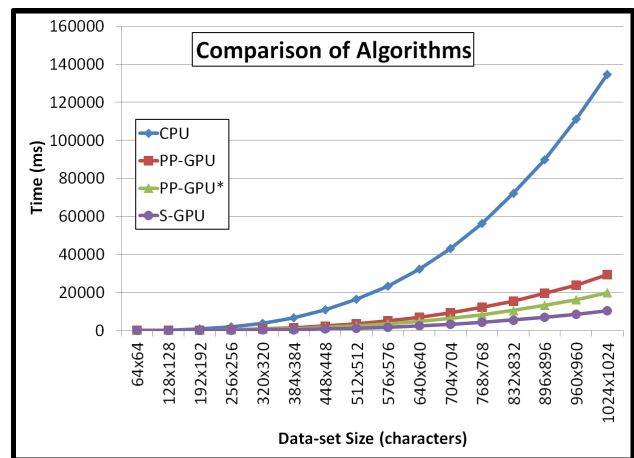


Fig. 12
COMPARISON OF ALGORITHMS CPU, PP-GPU(*), AND S-GPU:
 $L = 16, d = 4$

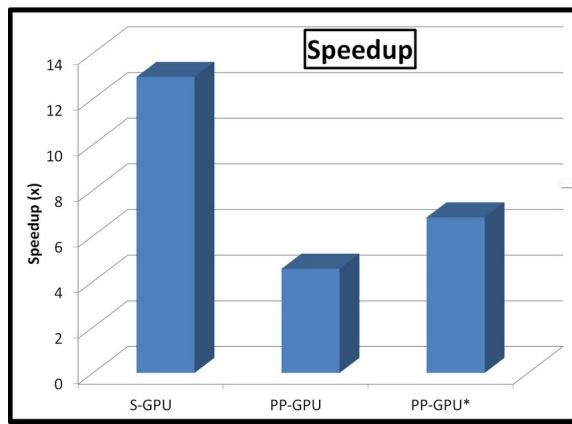


Fig. 13

OVERALL ALL SPEEDUP OF GPU ALGORITHMS:

$$L = 16, d = 4$$

Although PP-GPU attempted to preform the calculations of finding the hamming distances and chord errors in parallel, this action did not equate in much of a performance gain when compared to S-GPU. Its design was more fine grain than S-GPU, and thus needed more synchronization and collaboration to preform the calculation. The original design also facilitated the need for multiple kernel invocations which also contributed to the degradation in performance. The hardware fixed version PP-GPU* does provide a speed up of about 1.5x over its predecessor (PP-GPU) which had an overall speed up of 4.5x, and an speedup of 6.8x when compared to the CPU. S-GPU improved on this by utilizing a less fine grained approach to the problem allowing for the removal of key bottlenecks. In experiments, S-GPU ran 1.9x faster than PP-GPU* and achieved an overall speed up of 13x. It can be inferred from the results that more threads performing small jobs is not always conducive to better performance, rather a smaller number of threads performing slightly more work produced the best results in our experiments.

5. Related work

There has been much work done in areas closely related to CSSP. Problems such as multiple sequence alignment have experienced heavy research as of late.

Some have attempted to apply GPU algorithms to speedup multiple sequence alignment [9]. Although sequence alignment and CSSP are related they are not the same. CSSP is a more general problem. Perhaps ideas formed here can be adapted and applied to multiple sequence alignment tools such as Clustal. Perhaps ideas here can be applied to other areas in which forms of CSSP are represented.

Related work specific to CSSP has also been done. There have been many novel optimizations to approximation algorithms [1]. These algorithms generally perform much faster than an exact algorithm. The downside of course is that the results obtained are not verified. Our research focused on optimizing an exact algorithm, one who's result can be verified.

An approximate evolutionary algorithm for CSSP has been studied [8]. The algorithm attempts to change itself to make its results more accurate over time.

6. Conclusion

GPU devices can be used to efficiently solve the CSSP using parallel algorithms and GPU technology. For applications that require solving the CSSP or any of its relatives, efficient GPU algorithms can be developed that will permit computations that were previously too expensive.

Using parallel GPU algorithms and smart parallelization strategies we were able to greatly speedup the process of calculating the closest substring. The many existing applications that use a form of the CSSP can make use of GPUs to make their calculations faster. New applications could be built that previously could not due to the time required to calculate the closest string.

7. Future work

The algorithm can be expanded to work on larger data sets. The process of locating the lowest chord distance can be parallelized. The algorithm can be further optimized by using parallel minimization of the chord distances. This could result in an additional speedup for large data sets. The process of finding the chord distances can also be optimized using a parallel addition rather than the, sequential at worst, atomic add function included in the CUDA toolkit.

References

- [1] Bin Ma. "A Polynomial Time Approximation Scheme for the Closest Substring Problem (2000)" In Proceedings of the 11th Annual Symposium on Combinatorial Pattern matching, 2000, pp. 97-107
- [2] "NVIDIA CUDA C Programming Guide". Internet: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf November 9,2010 [March 21, 2011]
- [3] K. Lancot, M. Li, B. Ma, S. Wang, L. Zhang. Distinguish string search problems, Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 633-642, San Francisco, 1999
- [4] J. Sanders, E. Kandrot, CUDA By Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, 2010
- [5] D. B. Kirk, W. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010.
- [6] Ming Li, Bin Ma, and Lusheng Wang. "On The Closest String and Substring Problems" eprint ARXIV 2000.
- [7] Markus Chimani, Matthias Woste, Sebastian Bocker "A Closer Look at the Closest String and Closest Substring Problem" 2011 Workshop on Algorithm Engineering and Experiments (ALENEX) 2011.
- [8] Holger Mauch. "Closest Substring Problem - Results from an Evolutionary Algorithm," in Neural Information Processing, 1st ed., vol 3316. Ed. Nikhil Pal, Ed. Nik Kasabov, Ed. Rajani Mudi, Ed. Srimanta Pal, Ed. Swapan Parui, 2004, pp. 205-211.
- [9] Andrew Bellenir, Christian Trefftz, Greg Wolffe, "Graphics Processor Based Implementation of Bioinformatics Codes", 2008.

Achieving High Throughput Sequencing with Graphics Processing Units

Su Chen¹, Chaochao Zhang¹, Feng Shen¹, Ling Bai¹, Hai Jiang¹, and Damir Herman²

¹Department of Computer Science, Arkansas State University, Jonesboro, AR 72467, USA

²Department of Internal Medicine, University of Arkansas for Medical Sciences, Little Rock, AR 72205, USA

Abstract—High throughput sequencing has become a powerful technique for genome analysis after this concept was raised in recent years. Currently, there is a huge demand from patients that have genetic diseases which cannot be satisfied due to the limitation of computation power. Though several softwares are developed using currently most efficient algorithm to deal with various types of sequencing problems, the CPU seems to be too expensive to process endless data economically because CPUs are not designed adaptive for data parallel problem. The latest Fermi architecture released by NVIDIA provides considerable number of streaming processors, bigger size of register file and 1 MB cache, which makes it very competitive for data parallel processing. This paper tries a simple sequence alignment method on GPU and compared the real world performance between CPU and GPU. Experiment shows that GPU may have a good potential with similar problems.

Keywords: High Throughput Sequencing, Graphics Processing Unit

1. Introduction

Nowadays, people are paying more and more attention to health care and advanced devices are designed to analyze the samples from patients. When it comes to the molecular level, the data amount becomes extremely large, which needs more computational power to work on it. Recently, the emerging High Throughput Sequencing (HTS) technology [6], [7] shows bioinformaticists a way to deal with this problem better and many multithreaded programs such Bowtie [3], BWA [4] and SOAP2 [7], have been raised for practical use. However, for sequential CPUs, sequence alignment is somehow too easy to deal with, which makes it too expensive to use smart chips like CPUs. As NVIDIA released its new Fermi architecture which provide 512 cores in one chip and gigabytes of memory, GPU seems to have great potential in taking over this job and doing it faster and more economically.

In this paper, a simple way is proposed to do exact matching between massive DNA target fragments and mRNA reference sequences, and performance comparisons between its CPU and GPU version are discussed. The paper is organized as follows: Section 2 gives our method, including indexing and searching phases, to do sequencing. Section 3

discussed about the detailed designs considering architectures. In Section 4, we will discuss on the experimental results. Section 5 is the related work and conclusions will be drawn in Section 6.

2. Algorithm Design

The algorithm idea used in this paper comes from Burrows-Wheeler Transformation, which was first raised for data compression and was later developed to make an efficient index for sequence alignment. Fig. 1 illustrates how the original transformation works.

	acaacg\$		\$acaac g
	caacg\$a		aacg\$a c
	aacg\$ac		acaacg \$
acaacg\$	⇒ acg\$aca	⇒ acg\$ac a	⇒ gc\$aaac
	cg\$acaa		caacg\$ a
	g\$acaac		cg\$aca a
	\$acaacg		g\$acaa c

Fig. 1: Burrows-Wheeler Transformation

The concept of BWT is to make an index of reference sequence by hashing the elements within sequence to a special order, which will benefit later searching phase and reduce searching time complexity from $O(nlg(n))$ of brute-force method to $O(lg(n))$. Concrete implementation of BWT can be described as follows:

- 1) Put a "\$" at the end of reference sequence.
- 2) Copy the current sequence and shift the new sequence to right by 1 and put it below the last one for n times, given the original sequence length is n.
- 3) Sort the new generated block by the order of "\$", "a", "c", "g", "t" for each column.
- 4) Get the last column of the sorted matrix.

2.1 A New Indexing Method for Test

Inspired by BWT, we designed another way to make the index. Procedure of the new method is shown in Fig. 2. Next, we will explain it in a more detailed way.

- 1) We still add a "\$" at the end of the reference sequence.
- 2) Generate the same block as what BWT does. This time, we put order numbers for "a", "c", "g" and "t" separately for the first column.

- 3) In this approach, we only sort the first column of the matrix and make sure the small order numbers of “a”, “c”, “g” and “t” are on the top of the larger ones.
- 4) We get the last column as the new index.

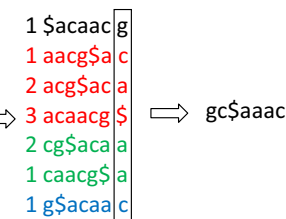
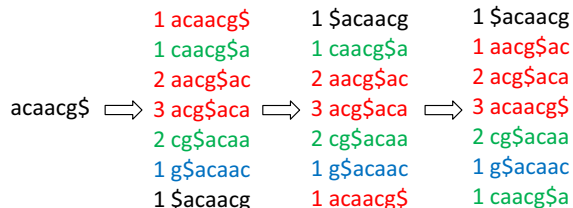


Fig. 2: New indexing method

2.2 Searching Algorithm

Search for: aac

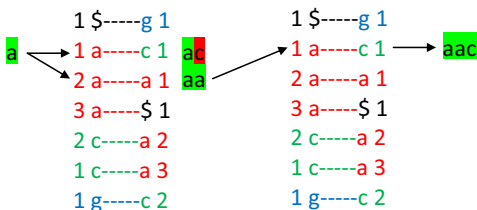


Fig. 3: Searching with the new index

The new proposed method has a brute-force searching nature, but by using the index well, several improvements can be achieved. The searching procedure is given in Fig. 3, which is very straightforward.

2.3 Making A Secondary Index

Now, we are going to talk about how to improve the performance of our searching algorithm. We can make a secondary index based on the first level index generated by the method mentioned above.

For the first column, since we will refer to the beginning and end of “a”, “c”, “g” and “t” many times, we can save some space and just record these position numbers for the four types of letter. This saves not only the searching time but also a lot of space for the index file. For the last column, since the “a”, “c”, “g” and “t” here are not clustered, we

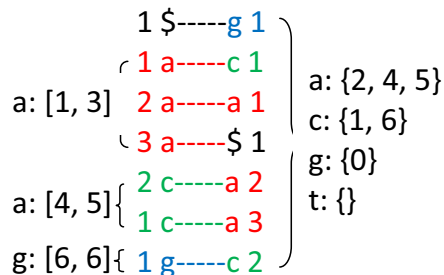


Fig. 4: Secondary index generation

can create four arrays for each of them and remember the occurrence positions in the last column for each element. This can prevent the searching algorithm from going to positions of wrong letters, for example, if we want “a”, we just go for 2, 4 and 5 positions in the last column and skip letters of other types.

Generally, though it does not fundamentally reduce the time complexity of the searching algorithm, this indexing method saves much unnecessary time by generating a simple index in $O(n)$ time, which is time-saving. In the experiment part, performances of CPU and GPU that we will be discussing about are based on this algorithm.

3. I/O Involved Program Design

3.1 Single-threaded Code Design for CPU

Since the indexing phase of our algorithm costs less time compared with the searching phase, in which an unpredictable number of target sequences will be throughput as inputs, we add the indexing time to the total searching time in this paper. Another important advantage of this is that we can save I/O time and load indices from hard-disk, which costs much more time than the indexing phase when the reference sequence file is very large. When we do everything in memory and never go to hard-disk, searching usually becomes faster. Fig. 5 illustrates how the data pertains our program flows between memory and hard-disk.

- 1) Load reference sequence file from hard-disk.
- 2) Generate index for reference sequence in memory.
- 3) Remove original sequence file from memory, only leave the index there.
- 4) Load the next target sequence file from hard-disk to memory
- 5) Do searching for the current batch of target sequences and save results.
- 6) Remove the first batch of target sequences.
- 7) Repeat 4) to 6) for all target files.

3.2 CUDA C code design for single GPU

Fig. 6 shows the procedure for a machine that has a CPU dealing with our problem. There are altogether nine steps of

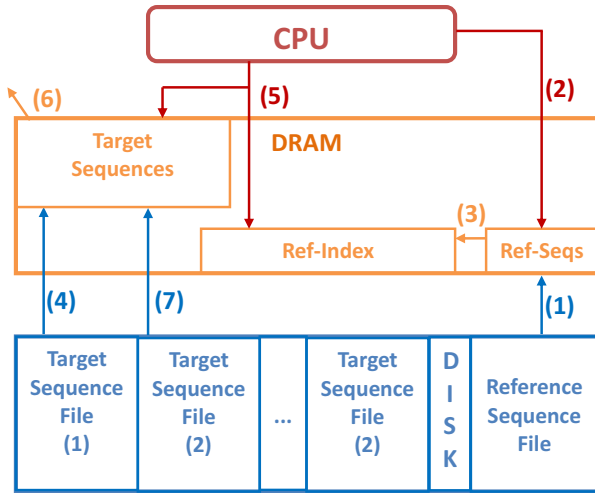


Fig. 5: Data locality control for CPU implementation

execution and data transfer for both indexing and searching phases, which will be explained more specifically next.

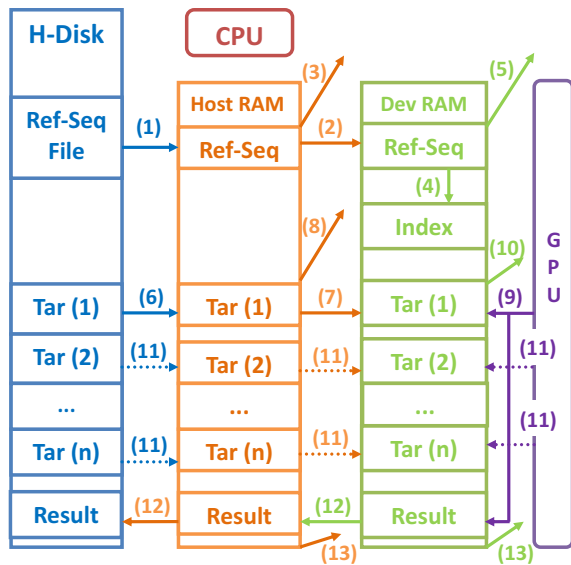


Fig. 6: Work and data scheduling for GPU implementation

- 1) Load reference sequence file from hard-disk to CPU memory.
- 2) Copy reference sequences from CPU memory to GPU memory.
- 3) Remove reference sequences from CPU memory.
- 4) Generate index for reference sequence using GPU.
- 5) Remove original sequence file from GPU memory, only leave index there.
- 6) Load a target sequence file from hard-disk to CPU memory.

- 7) Copy current batch of target sequences in CPU memory to GPU memory.
- 8) Remove present target sequences in CPU memory. Load the next batch of target sequences.
- 9) GPU does searching and save result in its memory.
- 10) Remove current batch of target sequences from GPU memory.
- 11) Repeat 6) to 10) for all target files.
- 12) Copy back result to CPU memory and save it to disk.
- 13) Remove results in GPU and CPU memory.

3.3 Noteworthy Differences between CPU and GPU implementations

- 1) The GPU one has an initializing time for the first booting of the device, usually taking up to 2-3 seconds, where CPU one does not. So for small cases that can be run very fast on CPUs, GPUs have no advantage.
- 2) Data transfer time between host and device memory should be considered since data amount in our case is usually very large.
- 3) GPUs can do simple calculations very fast if programs are designed well, so indexing and searching phase can also be considered to do in GPUs, if the data transfer time can be ignored. If the indexing time requires only a little, there is no much need to do it in GPUs. Searching phases usually can be taken well on GPUs since target sequence numbers are always very large. Acceleration rate of dozens to hundreds can be expected for the searching phase if GPUs are adopted.

4. Experimental Results

Sequential code was written in C and tested on a machine with two Intel Xeon E5504 Quad-Core CPUs (2.00GHz, 4MB cache), where GPU code was written in CUDA C and tested on the same machine with two GPUs of NVIDIA Tesla 20-Series C2050. In the following part, performance comparison between these two will be given and speedup rate for GPU will be calculated out. Also, time proportion for each part of whole algorithm on CPUs and GPUs will be illustrated and discussed separately.

4.1 CPU vs. GPU Searching Time

Block sorting is the most time consuming part in making index for reference strings. Fig. 7 gives the relational curves about time cost and combination number of reference strings (one reference string length = 3,000).

From Fig. 7 we can see that for the algorithm proposed in this paper, searching time takes a big portion of total execution time on the CPU side while on the GPU side, it takes relatively smaller portion. This is because GPU runs much faster on the searching part compared to CPU, so given the I/O and data transfer time changes proportionally as the target sequence number increases, GPU saves more absolute time as the problem scale becomes larger and larger.

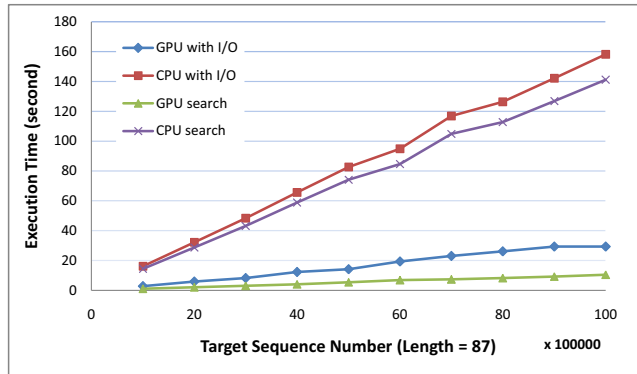


Fig. 7: CPU & GPU timing with and without I/O

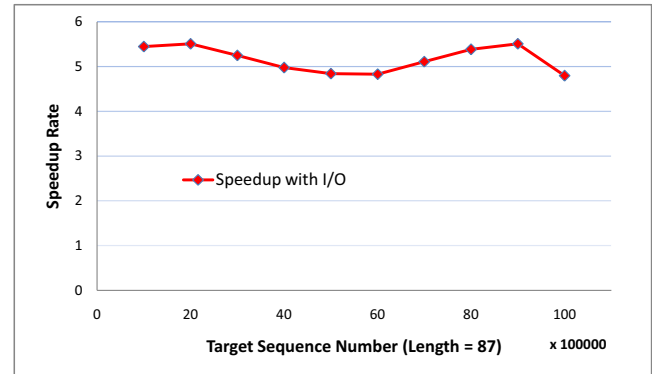


Fig. 9: GPU speedup rate with I/O

4.2 Speedup on GPU

Fig. 8 and Fig. 9 illustrate two speedup curves about the pure searching time and searching time with I/O and data transfer. We can see that for pure searching algorithm, the GPU one can beat the CPU one for up to 14 times, where about 5 times speedup can be achieved when I/O and data transfer is taken into consideration. Actually, since the algorithm is not ultimately optimized, there should still be potential for GPUs to speed up this problem.

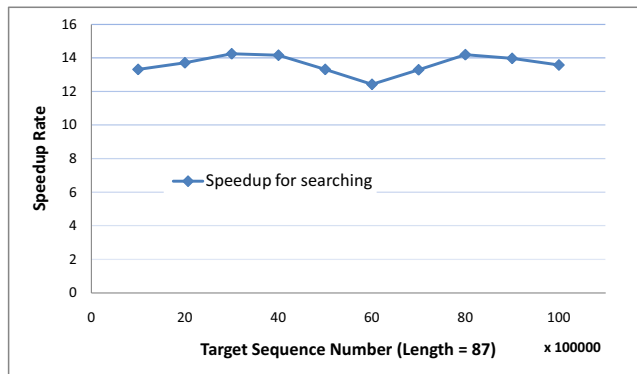


Fig. 8: GPU speedup rate without I/O

4.3 Overhead Breakdown with CPU & GPU Approaches

1) I/O from hard-disk

For both CPU and GPU implementations, this part should take the same time, which is inevitable. The bandwidth from hard-disk to memory has always been a bottleneck for similar problems. However, if we are not using the local hard-disk but using InfiniBand to load data from remote database in parallel, the performance for both CPU and GPU once can be improved, where GPU one might benefit more

because it processes data much faster than CPU one and need more data in a given time to meet its stronger computation power.

2) Data transfer between host and device memory

Currently, NVIDIA GPUs are using PCIe bus to transfer data from and back between host and device memory, whose capacity is up to 4GB/s for one way transmission and 8GB/s for two way. This speed usually can satisfy GPU's computation power and will not be a bottleneck for now. A noteworthy thing about this is that asynchronous memory copy technique should be used when target sequence is too large to load for once by GPU memory. Asynchronous copy between host and device memory can overlap with GPU computation, so either copy or computing time can be hidden by this overlapping. Which portion will be hidden depends on their time costs.

3) Time for indexing

For the algorithm presented in this paper, indexing time can nearly be ignored since I/O and searching time dominate. However, in real applications, such as BWT, indices are usually made more efficient to use. But it also takes more time on indexing and the overhead cannot be ignored. In that case, indexing time should also be considered as an important portion of the whole system.

4) Time for searching

This portion of time relies on many factors including indexing efficiency, I/O speed, choose of device and task partitioning design. Basically, more efficient indexing can reduce searching time whereas higher I/O speed can positively influence the performance. For device choosing, we can say GPU is better than CPU from the angle of economy since it provides more powerful tools for searching. However, whether a partitioning design is good or not is hard to tell if we just look at the surface of a specific problem. Calculations should be carefully done to find out the optimum selection for it.

5. Related Work

RNA sequencing was one of the earliest forms of nucleotide sequencing. The major landmark of RNA sequencing is the sequence of the first complete gene and the complete genome of Bacteriophage MS2, identified and published by Walter Fiers et al. in 1972[8] and 1976[2].

In late 2000 decade, high-throughput sequencing (HPS) emerged. Li R (2008, 2009) proposed several papers about BWT applications on short read alignment [6], [7]. Li H (2008, 2009) [5], [4] and Langmead (2009) [3] also published several works about memory-efficient alignment. In recent years, several alignment programs such as Bowtie [3], BWA [4] and SOAP2 [7] were released.

In 2009, Sinnott-Armstrong et al. presented a paper about accelerating epistasis analysis in human genetics with Nvidia GeForce GTX-280 and PyCUDA programming tool [9]. Nicholas et al. (2011) made a real-world performance comparison of SNPrank across programming platforms such as Python, Java and Matlab, and hardware environments: single threaded, multiple threaded and GPU, where GPU languages are restricted to Matlab and Python [1] and GPU brand is Nvidia Tesla-M1060. They declared for small cases, CPU always performs better because of the data transfer to and from device memory.

6. Conclusions and Future Work

This paper proposes a way to implement fast sequence alignment on the latest version of NVIDIA GPU. From the experimental result, we can see that GPU speeds up more on the searching phase compared with CPU but delays a constant length of time on its necessary data transfer phase. This feature of GPU manifested that it has a good potential for high throughput sequencing. If the bandwidth bottleneck of loading data from hard-disk can be improved, the performance still has a great potential to keep growing; where for single threaded CPU, the computation power may not guarantee that.

In future, we will try to parallelize the most advanced sequence alignment algorithm on GPU and keep investigating the GPU's capability on more applications that receive urgent concerns from medical and biological fields.

References

- [1] Nicolas A. Davis, Ahwan Pandey, and B. A. McKinney. Real-world comparison of cpu and gpu implementations of snprand: a network analysis tool for gwas. *Bioinformatics*, 27(2):284–285, 2011.
- [2] W Fiers, R Contreras, and F Duerinck. Complete nucleotide sequence of bacteriophage ms2 rna: primary and secondary structure of the replicase gene. *Nature*, 260:500–507, 1976.
- [3] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, 10(3), 2009.
- [4] H Li and R Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.
- [5] H Li, J Ruan, and Durbin R. Mapping short dna sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):1851–1858, 2008.
- [6] R. Li. Soap: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.
- [7] R. Li. Soap2: an improved ultrafast toll for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.
- [8] Jou W. Min, G. Haegeman, M. Ysebaert, and Fiers W. Nucleotide sequence of the gene coding for the bacteriophage ms2 coat protein. *Nature*, 237(3069654):82, 1972.
- [9] Nicolas A Sinnott-Armstrong, Casey S Greene, Fabio Cancare, and Jason H Moore. Accelerating epistasis analysis in human genetics with consumer graphics hardware. Technical report, Dartmouth Medical School, NH, USA Politecnico di Milano, Milano, Italia, 2009.

Optimization of a single seam removal using a GPU

Rok Češnovar, Patricio Bulić, Tomaž Dobravec

University of Ljubljana, Faculty of Computer and Information Science, Tržaška c. 25, Ljubljana, Slovenia

Abstract—*In this paper we consider the problem of implementing and optimizing the Seam Carving algorithm on graphics processing units. Seam Carving is a content-aware image resizing method proposed by Avidan and Shamir. In order to use their proposed method in real-time application, a pre-processing step is needed. While some other papers propose real-time resizing by changing the original Seam Carving method, this paper focuses on optimizing the basic single-seam method, for use on CUDA GPUs. To our best knowledge none has focused on optimizing this single-seam method so that it would best fit the GPU architecture.*

Keywords: Seam Carving, CUDA, shared memory, GPU

1. Introduction

The increasing number of different-sized displays has resulted in the increased demand for image resizing. Traditional image resizing techniques (image scaling, cropping) are known to have some deficiencies, the biggest being obliviousness to the image content. Thus changing the aspect ratio with these techniques causes significant distortions.

Avidan and Shamir [1] proposed a new, content-aware image resizing method, called Seam Carving. This method resizes the image by adding or removing seams. A seam is a 8-connected path of low energy pixels crossing the image from top to bottom, or left to right, while only one pixel in a row or column, respectively, can be a part of the seam. In order to determine which pixels belong to the optimal seam, dynamic programming is used. This proposed method can be divided in 4 steps:

- 1) energy analysis
- 2) cumulative minimum energy computation
- 3) backtracking for optimal seam determination
- 4) seam removal

In order to reduce/enlarge the image in any dimension for k pixels, all of the above steps have to be repeated k -times in order to achieve the best results. Repeating all the steps is very time consuming, thus real-time application can not be achieved this way. In [1] the authors provided a solution for this problem, using a slow pre-processing step. With this step we determine the sequence in which the seams can be removed and the information about each seam. As we can imagine this step is very memory consuming.

A number of papers were published on the topic of real-time content-aware image resizing. In [3] the authors proposed an efficient improvement of the seam carving method, by

detecting local and global seams. This way multiple seams can be removed in each step, instead of the before mentioned repetitions of single seam removal. In [4] the authors also propose a more efficient approach to seam based content-aware image resizing, searching the seams through establishing the matching relation between adjacent rows or columns. In this paper, instead of focusing on changing the seam carving algorithm, we focused on and optimized implementation of the original single seam removing method on GPUs. The optimization is based on the properties of the GPU we used. We present the results of this implementation and the results of the optimization steps. The method was implemented on graphics processing units with the CUDA architecture and programming model.

CUDA(Compute Unified Device Architecture) is an open parallel architecture and a programming model, developed in 2007 by the nVIDIA corporation[2]. It gives the developers a set of abstractions that enable expressing fine-grained and coarse-grain data and task parallelism. CUDA provides us with the possibility to run our algorithms massively parallel, thus shortening the execution time and reducing the CPU load.

This paper is organized as follows. In Section 2 we give an overview of the Seam Carving method, in order to give the reader a better understanding of our implementation. In Section 3 we provide the reader with some basic information on the CUDA programming model and architecture in order to give the reader a better understanding of the presented results and conclusions. In Section 4 we propose the method for parallelizing and optimizing of single seam removal explained in Section 2. In Section 5 we present the measurement results of the algorithm on our testing equipment. And finally in Section 6 we outline some conclusions and give the proposal for the future works.

2. Seam Carving

In this section we will give a brief overview of the seam-carving method proposed in [1]. This should give the reader a better understanding of the implementation proposed in Section 4. All of the definitions in this section are based on [1].

The seam carving method is used for image resizing. Content-aware resizing is done by removing unnoticeable pixels from the image. In order to determine which pixels are the least important in our image we use an energy function. An energy function determines the importance of

each pixel in the image. For the purpose of this paper we implemented 2 energy functions with their main difference being in computational complexity. A more complex energy analysis was done with the Sobel operator 3x3:

$$G_x = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A, G_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A \quad (1)$$

$$e(i, j) = \sqrt{G_x^2 + G_y^2}, \quad (2)$$

where $e(i, j)$ is the energy of the pixel in the i -th row and j -th column and A is a matrix representing the image. The less complex energy analysis was done with the following function:

$$e(i, j) = |I(i, j) - I(i, j + 1)|/3 + |I(i, j) - I(i + 1, j)|/3 + |I(i, j) - I(i + 1, j)|/(3\sqrt{2}) \quad (3)$$

where $I(i, j)$ is the value of the pixel at (i, j) . Lets assume we want to reduce the image width. In order to keep the rectangular shape of the image we need to remove the same number of low energy pixels in each row. If this would be the only constraint, the resulting image could contain the zigzag effect. This leads to the definition of a vertical seam. *Definition 1:* (Vertical seam) Let I be an $n \times m$ image. A vertical seam is, as shown below, a 8-connected path of pixels from top to bottom of the image.

$$s^x = \{s_i^x\}_{i=1}^n = \{(x(i), i)\}_{i=1}^n, s.t. \forall i, |x(i) - x(i - 1)| \leq 1, \quad (4)$$

where x is a mapping $x : [1, \dots, n] \rightarrow [1, \dots, m]$. Reducing the image width can now be seen as removing the optimal seam, formalized in the following definition:

Definition 2: (Optimal seam) Let $e(i, i)$ be the energy of the pixel in i -th row and j -th column. An optimal seam is, as shown below:

$$s^* = \min_s \sum_{i=1}^n e(i, j) \quad (5)$$

where x is a mapping $x : [1, \dots, n] \rightarrow [1, \dots, m]$. This step is done with dynamical programming, which is divided into 2 steps. First we need to compute the cumulative minimal energy M of each pixel.

Definition 3: (Cumulative minimal energy) For pixels from the second to the last row cumulative minimal energy M is:

$$M(i, j) = e(i, j) + \min(M(i - 1, j - 1), M(i - 1, j), M(i - 1, j + 1)), \quad (6)$$

while $M(1, j) = e(i, j)$. The pixel with the lowest M in the last row represents the end of the vertical seam. In the last step we backtrack from this pixel to the top of the image, the path of the backtrack representing the optimal seam. To reduce the image width, we only need to remove these pixels from the image.

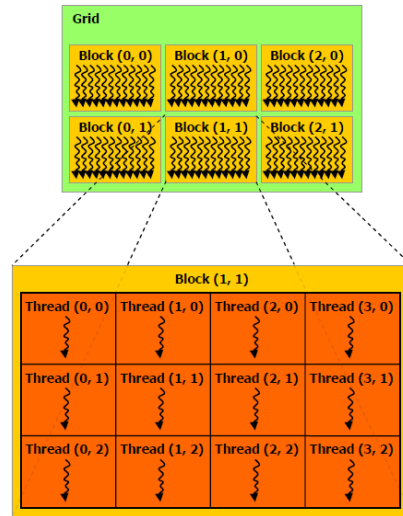


Fig. 1: Grid of thread blocks [2]

As mentioned before, in order to reduce the width/height for k pixels all the steps including the energy analysis need to be repeated.

3. CUDA programming model and architecture

In this section we will give an overview of the CUDA programming model and architecture. We will give the basic principles of writing a CUDA program and the architectural features on which our implementation and optimization was based on. Only the most essential information is given, but enough to understand the following implementation and results. More information on the topic of this section is given in [2].

A CUDA program is separated in 2 parts, CPU code and GPU kernels. A kernel is a C function that, when executed, is called N times in parallel by N different CUDA threads. These threads are organized in one, two or three-dimensional logical blocks. Blocks are furthermore organized in 1 or 2-dimensional grids. The organization can be seen on figure 1.

Each thread has a unique pair of ID variables (*threadIdx*, *blockIdx*). Both of these variables are, due to the multi-dimensional logical structure of threads, component vectors. The organization of threads depends on the size and structure of the data and the problem.

Each of the multiprocessors on a GPU is given a number of thread blocks to execute. The multiprocessor partitions these blocks in warps. A warp is a group of 32 parallel threads. Each warp contains threads of consecutive, increasing *threadIDs*, and executes one common instruction at a time for all threads in a warp.

It is important to understand that we can only synchronize the execution of threads in the same block, while synchro-

nization of threads in different blocks can only be achieved through loading a new kernel.

While focusing on the thread organization is important for achieving the best results, optimizing the memory access is, as we will see in section 5, the main task in optimizing any CUDA program.

Each thread can access different types of memory. In our implementation the threads are accessing global and shared memory. Every thread can communicate with any other thread using global memory, while only threads in the same block can communicate through shared memory.

Global memory resides in the device memory, while shared memory is on-chip. Therefore shared memory access is faster, as long as there are no bank conflicts. When using global memory we have to focus on maximizing the coalescing of memory accesses, while bank conflicts are our main focus when using shared memory. The importance of coalescing of memory accesses is closely related to the type of GPU we are using.

A global memory request for a warp is split into 2 half-warps. Memory accesses of threads in a half-warp can be coalesced in one memory transaction, if proper requirements are met. The requirements for coalescing to a single transaction depend on the GPU architecture. Older CUDA enabled GPUs supported coalescing only if consecutive threads accessed consecutive addresses ($k - th$ thread accesses $k - th$ address), while newer GPUs coalesces the memory accesses even if the addresses are permuted.

4. Implementation

In this section we will propose the implementation and optimization of the algorithm for graphics processing units that support CUDA.

The implementation is separated in 4 parts: energy analysis, minimal cumulative energy computation, backtracking and seam removal. For each of the first 4 parts a separate kernel was written. This is due to the different thread organization needed in every part of the algorithm and due to the need for synchronization of all threads after each step.

Before we can start the computation we need to transfer the matrix representing our image to the GPU device memory. Then we can start with our first step.

4.1 Energy analysis

First lets look at an unoptimized kernel for energy analysis (Listing 1). Because images are two-dimensional structures, we define a two-dimensional logical structure of threads. Each block consists of $p \times p$ threads and the size of the grid equals to $\lceil \frac{m}{p} \rceil \times \lceil \frac{n}{p} \rceil$, where n is the height and m is the width of the image. This way we can assign a thread for each pixel of the image.

In the kernel for this step, we first need to determine the pixel to which the thread was assigned to. This is done using the *threadIdx* and *blockIdx* variables (lines 5-7 in Listing 1.

All that is left, is the computation of the energy value. In this kernel, all memory accesses are made to the global memory. The listing shown below is a kernel for energy analysis using the energy function shown in equation 3.

```

__global__ void Energy(int* I, float* O,
    int width, int height){

    int j =blockIdx.x*blockDim.x+threadIdx.x;
    int i =blockIdx.y*blockDim.y+threadIdx.y;
    int id=i*width+j;

    O[id]=(abs(I[id]-I[id+width])
        +abs(I[id]-I[id+1])+abs(I[id]
        -I[id+width+1])/sqrt(2.0))/3;
}

```

Listing 1: Energy analysis kernel without the use of shared memory

Due to the large number of global memory accesses we look at the possibility of reducing them by using shared memory. In this approach, each thread has to transfer the value of its assigned pixel to the shared memory, before computing the energy. In order for each thread, to have all the values needed for energy analysis in its shared memory, the block size needs to be enlarged. When using the function from equation 3 the new block size is $(p+1) \times (p+1)$. When using the Sobel 3x3 operator, the new block size is $(p+2) \times (p+2)$. The newly padded threads are only used for data transfer. After the transfer of data from global to shared memory, synchronization is needed, in order to ensure all data was transferred. After that, we can compute the energy using the values from the shared memory. In listing 2 the kernel for energy analysis using shared memory is shown. It should be noted that, for transparency reasons, parts of code used to check for boundary conditions was left out.

```

__global__ void EnergySH(const int* I,
    float* O, int width, int height){

    int j =blockIdx.x*blockDim.x+threadIdx.x
        -blockIdx.x;
    int i =blockIdx.y*blockDim.y+threadIdx.y
        -blockIdx.y;
    int id=i*width+j;

    int r=threadIdx.y;//row number
    int c=threadIdx.x;//column number
    __shared__ int Is[p][p];

    Is[r][c]=I[id];

    __syncthreads();
    O[id]=(abs(Is[r][c]-I[r+1][c])
        +abs(Is[r][c]-I[r][c+1])
        +abs(Is[r][c]-I[r+1][c+1])/sqrt(2.0))/3;
}

```

Listing 2: Energy analysis kernel with the use of shared memory

Besides the fact that shared memory access is faster it also allows us to remove a number branches used for checking the boundary conditions. As stated in [2], in order to reduce the execution time, the number of branches has to be minimized.

4.2 Minimal cumulative energy computation

The second step of the algorithm is the computation of minimal cumulative energy. The size of the block for this step of the algorithm is $r \times 1$, and the number of blocks equals to $\lceil \frac{m}{r} \rceil \times 1$. Due to the fact that, for computing the values in the k -th row, all values in the $(k-1)$ -th row need to be computed beforehand, synchronization between all the threads is needed. The only way to achieve this is by repeatedly loading the kernel for each row. We start by loading the kernel for the second row, and continue all the way to the last row. In listing 3 the kernel for this step is shown.

```

_global__ void cumulative(float* O,
int width, int height, int row){
int j=blockIdx.x*blockDim.x+threadIdx.x;
int id=i*width+j;

int idSouth=id+width;

int min=O[idSouth];
if(O[idSouth-1]<min) min=O[idSouth-1];
if(O[idSouth+1]<min) min=O[idSouth+1];

O[id]=O[id]+min;
}

```

Listing 3: Kernel for minimal cumulative energy computation

Shared memory can also be used in this step. The implementation of shared memory resembles to the one in the energy analysis. The block needs to be, for the same reason as before, padded with 1 thread on each side of the block and synchronization is also needed. The only difference is that the structure of the shared memory is one-dimensional.

4.3 Backtracking

For this step of the algorithm a single thread is used. First it finds the minimal value in the last row. This is the end of our optimal seam. Then it continues to select the pixels for the optimal seam all the way to the first row. In every row the thread finds the minimal value of all 8-neighbours of the pixel selected in the previous row. The pixel with the mentioned minimal value is selected for the optimal seam.

4.4 Seam removal

In the final step of the single seam removal we define the block size to be $p \times p$ and the number of blocks equals $\lceil \frac{m}{p} \rceil \times \lceil \frac{n}{p} \rceil$. Each thread is assigned a pixel. Threads not

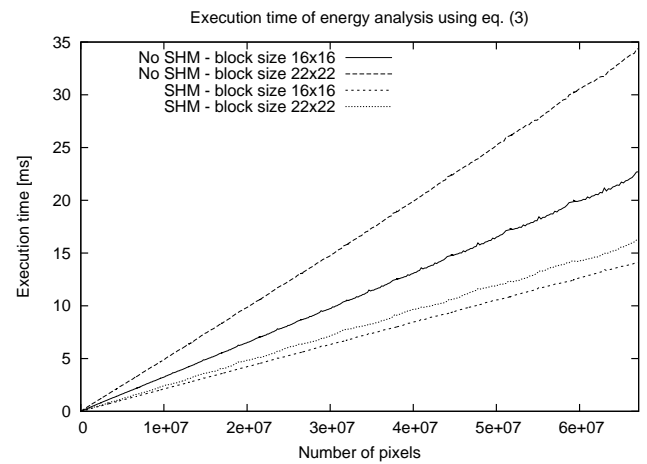


Fig. 2: Execution times of energy analysis using eq. 3

assigned to the seam pixels transfer the pixel values to the new matrix, representing our result. If the assigned pixel is located right of the seam, the thread moves the pixels value to the column left of its location in the current matrix, other pixels are assigned to the same position in the new matrix as in the input one.

A new, result matrix, is needed, in order to prevent this step from dependency issues.

After the seam is removed we transfer the new matrix, representing our result, back to the main memory of our computer.

If we wanted to remove more seams, we would use the result matrix as our new input. In this case no transfer to global memory is needed between steps.

5. Results

In this section we will present the results of our execution time measurements. We ran our parallel version of the seam carving method on a Tesla C1030 GPU. The consecutive version was ran on a Intel i5 CPU.

The first measurement test we ran, was to determine what is the optimal way of computing the energy analysis. In Figure 2 execution times of the energy analysis, using the energy function from equation 3, is shown. Figure 3 represents the execution times when using the Sobel 3x3 operator, which is a more complex energy analysis and demands a greater number of memory accesses. Based on this measured times we came to the following conclusions.

When doing the energy analysis with the energy function from eq. (3) the optimal block size is 16×16 threads. This is mostly due to the size of the half-warp being 16, thus optimal requirements for coalescing of memory accesses are met. Furthermore, the use of shared memory reduces the execution time in this energy analysis for approximately 38 percent.

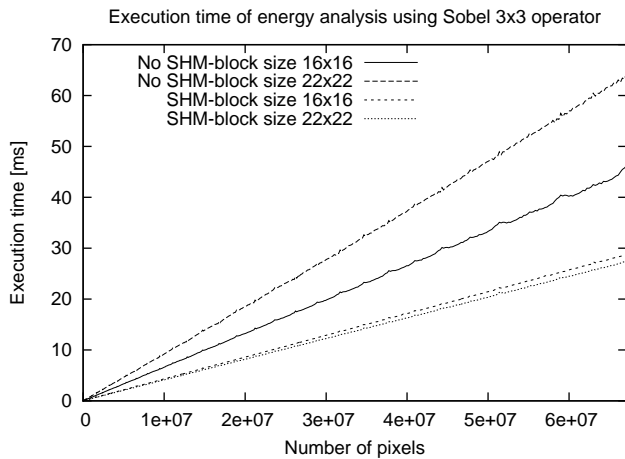


Fig. 3: Execution times of energy analysis using Sobel 3x3

When using Sobel 3x3 operator with shared memory the optimal block size is 22×22 , when not using shared memory the optimal block size is 16×16 threads. By using shared memory we reduce the execution time by 40 percent. The reason for the optimal block size being larger than in the previous energy analysis, is the increased number of padded threads per block.

By increasing the block size, we reduce the ratio between padded threads and threads that compute the energy values. The block size 22×22 is the largest square-sized block of threads (maximum number of threads is 512), thus provides us with the smallest ratio between the before-mentioned threads.

The next measurement tests were done to determine the optimal way of computing the minimal cumulative energies. We found that the optimal size of the block for this step is 256×1 threads. If the block size is bigger, not all multiprocessors are occupied, and if it is smaller, the streaming processors inside a multiprocessor are not optimally occupied.

We also found that the use of shared memory is not recommended. If we compare this step to the energy analysis, where shared memory proves to be useful, we can state that, the number of boundary conditions and global memory accesses in this step is smaller, thus the advantages of reducing their number does not overcome the disadvantage of synchronizing the threads.

Now that we found the optimal way of executing the method on a GPU, we can compare the execution times of optimized and unoptimized version of the parallel algorithm to the execution times of the concurrent algorithm, run on a CPU. In figure 4 execution times for all three types are shown. In figure 5 the speedup factor when using GPU is shown. We can see a significant reduce of execution time when using GPU. For example, at the image size 1024 the CPU needs 56ms to remove the seam, while the optimized version of the GPU algorithm needs 11.5ms to the same.

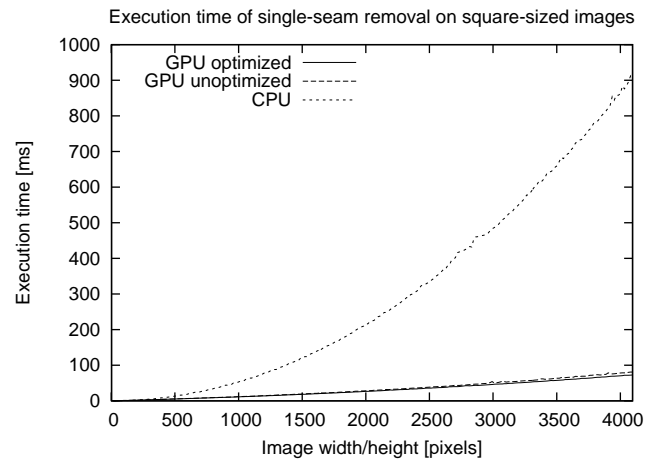


Fig. 4: Execution times of Seam Carving using CPU and GPU

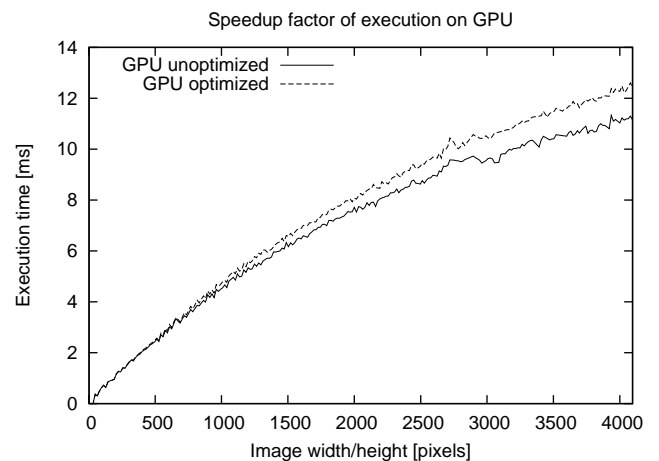


Fig. 5: Speedup factor when using GPU

The unoptimized GPU version of the algorithm needs 12ms. Compare this to 0.9s, 81ms and 72ms which are the execution times of CPU, unoptimized an optimized GPU algorithms at the image size 4096×4096 pixels. We can see a 10 percent reduction of the execution time with optimizing the algorithm for GPU use.

The speed-up factor rises up to over 12.5 at the image size of 4000×4000 pixels, when using the optimized version, and up to 11, when using the unoptimized version of the algorithm at the same image size.

Both GPU implementations are faster than the CPU one when the image size is bigger than 150×150 pixels.

6. Conclusions

In this paper we described the optimization of single-seam removal using GPU. We have shown that shared memory should be used in order to reduce the execution time of the energy analysis and that its not useful to apply it to other steps of the algorithm. We have shown that the use of shared memory proves to be more useful when the energy analysis is computationally more complex and demands more memory accesses. Furthermore we have shown that, when writing an algorithm for GPUs, we need to focus on the block size we define, in order to get the best results.

Future research will consider the possibility of parallelizing the backtracking step of the algorithm and applying the same methods of optimization to the works shown in [3],[4] and [5].

References

- [1] Avidan S., Shamir A. *Seam Carving for Content-Aware Image Resizing*. ACM Trans. Graph. Vol 26., No. 3, 2007
- [2] NVIDIA Corporation. *CUDA C Programming Guide Version 3.1.1*. June 2010, CUDA Toolkit
- [3] Chen-Kuo C., Shu-Fan W., Yi-Ling C., Shang-Hong L. *Fast JDN-Based Video Carving with GPU Acceleration for Real-time Video Retargeting*.
- [4] Huang H., Fu T., Rosin P., Qi C. *Real-time content-aware image resizing*. June 2010, CUDA Toolkit
- [5] Rubinstein M., Shamir A.,Avidan S. *Improved seam carving for video retargeting*. ACM Trans Graph(SIGGRAPH), 2008, 27(3): 1-9

An Experiment in Parallelizing the Fast Fourier Transform

Timothy W. O'Neil, Ameen B. Mirza and Dale H. Mugler
The University of Akron

Abstract - We present the parallel implementation of two new algorithms developed for the discrete cosine transform. These algorithms support the new interleaved fast Fourier transform method. Our techniques were realized using the MPI standard library and executed on a variety of equipment for comparison. The results indicate a promising fresh direction in the search for efficient ways to compute Fourier transforms.

1. Introduction

The Fourier transform remains one of the greatest algorithmic discoveries in history. Its ability to convert the time domain to the frequency domain has wide-ranging applications in science and engineering, including digital signal processing, voice recognition and image processing. Unfortunately, its use has been hampered by its computational expense as a rule. It is the ongoing search for effective implementations that motivate this work.

We have attacked the problem in two ways: refine the traditional fast Fourier transform (FFT) algorithm, and implement the refinement in parallel for expedience. Parallel implementations of the classic FFT are quite common, particularly those using the Message Passing Interface (MPI) standard library [1,2]. Our purpose is to investigate the parallel implementation of a new interleaved FFT algorithm [3,4] which heavily relies on the discrete cosine transform (DCT). Our early results seem to indicate improved computing performance for large data sets, leading to the promise of more efficient PDE solutions, compression tools and MRI scans.

In the next section, we briefly review the formal mathematical definitions and theory behind Fourier transforms. Then we describe the computing

equipment used, the two DCT algorithms (gg90 and lifting) developed, our parallel FFT designs and our testing results. Finally we conclude and point to future work.

2. Background

The mathematical theory behind the discrete cosine and fast Fourier transforms is common in the literature and only briefly reviewed here for completeness. If $x(t)$ is a periodic time function, it can be decomposed into a series of sinusoidal waveforms of varying frequencies and amplitudes. In general, $x(t) = \int_{-\infty}^{+\infty} X(f) e^{2\pi i f t} df$ where $e^{iu} = \cos u + i \sin u$ and $X(f) = \int_{-\infty}^{+\infty} x(t) e^{-2\pi i f t} dt$ is the Fourier transform of $x(t)$, a continuous function of frequency [1].

However, computers lend themselves to discrete rather than continuous math, leading us to sample the frequency at intervals and estimate this equation via summation. Specifically, choose N large enough so that $[-N/2, N/2]$ contains the interval where the n^{th} sample of $x(t)$ is non-zero. Thus the n^{th} series coefficient is $X_n = \int_{-N/2}^{N/2} x(t) e^{-2\pi i n t / N} dt = X\left(\frac{n}{N}\right)$ since $x(t)$ is zero outside the range in question. As N increases the coefficients more closely model the Fourier transform, so that under appropriate conditions the Fourier series of $x(t)$ equals $x(t)$ itself. In other words, $x(t) = \frac{1}{N} \sum_{n=-\infty}^{+\infty} X_n e^{2\pi i n t / N}$, a derivation also commonly referred to as the Fourier series. In reality it is impractical to collect an infinite number of frequency samples, leading us to sample $x(t)$ at N points and derive the discrete Fourier transform (DFT): $X_k = \frac{1}{N} \sum_{n=0}^{N-1} x_n e^{-2\pi i n k / N}$ where $x_k = \sum_{n=0}^{N-1} x_n e^{2\pi i n k / N}$ for $k = 0, 1, \dots, N-1$ [1,2].

If $x(t)$ is defined only for $t \geq 0$, we can define a new function $y(t) = x(t) \cdot (t \geq 0) + x(-t) \cdot (t \leq 0)$ and see that the continuous Fourier transform becomes

$$\begin{aligned} Y(f) &= \frac{1}{\sqrt{2\pi}} \left(\int_0^{+\infty} x(t) e^{-2\pi f t i} dt + \int_{-\infty}^0 x(-t) e^{-2\pi f t i} dt \right) \\ &= \frac{1}{\sqrt{2\pi}} \int_0^{+\infty} x(t) \left(e^{-2\pi f t i} + e^{2\pi f t i} \right) dt \\ &= \sqrt{\frac{2}{\pi}} \int_0^{+\infty} x(t) \cos(2\pi f t) dt \end{aligned}$$

and thus $Y(f)$ is the *Fourier cosine transform* of $x(t)$. It can be faster for some applications due to its use of only real values. Using the same kind of logic as before, we can derive a preliminary version of the

discrete cosine transform: $Y_k = \sqrt{\frac{2}{N}} \sum_{n=0}^N x_n \cos\left(\frac{\pi k n}{N}\right)$

[3]. Note that this is something of a primitive version of the DCT. There are actually eight standard types [6], with a variation on DCT-IV used in the interleaved FFT algorithm [4].

Finally, we note that the DFT can be rewritten to separate terms with even and odd indices:

$$\begin{aligned} X_k &= \frac{1}{N} \left(\sum_{n=0}^{N/2-1} x_{2n} e^{-2\pi i (2n)k/N} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-2\pi i (2n+1)k/N} \right) \\ &= \frac{1}{2} \left(\frac{1}{N/2} \sum_{n=0}^{N/2-1} x_{2n} e^{-2\pi i n k / N/2} + \frac{e^{-2\pi i k / N}}{N/2} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-2\pi i n k / N/2} \right) \end{aligned}$$

which we can summarize as $X_k = \frac{1}{2}(X_{\text{even}} + e^{-2\pi i k/N} X_{\text{odd}})$. Thus the DFT reduces to the problem of evaluating two equations each with half the degree at the squares of the roots of unity and combining the results. This decomposition is the basis of the classic Cooley-Tukey *fast Fourier transform* algorithm, which recursively applies this idea and permits calculation of the DFT in $O(N \log N)$ time rather than $O(N^2)$ time [5].

3. Methods and Implementation

The primary contribution of this work involves the implementation of a new interleaved FFT algorithm which is heavily reliant on the DCT, then testing it on data sets of varying sizes for execution times.

3.1. Available Hardware and Software Configurations

For completing our experiments, we had access to two parallel computers. The first was the cluster in the University of Akron (UA) Computer Science department. It consists of 46 3 GHz Intel® Pentium® D CPUs for compute nodes, each with access to 2 GB local RAM. The nodes are connected by dual gigabit networks on private switches for cluster communication. One network is used only for diskless operations, the other only for MPI traffic. A single front node is used for user access.

We also conducted experiments using the IBM Cluster 1350, named “Glenn”, at the Ohio Supercomputer Center (OSC) in Columbus for some experiments. At the time, Glenn was configured with 877 System x3455 compute nodes (dual socket, dual core 2.6 GHz Optrons with 8 GB local RAM and 48 GB local disk space), 88 System x3755 compute nodes (quad socket, dual core 2.6 GHz Optrons with varying amounts of local RAM (16, 32 or 64 GB) and disk space (218 GB or 1.8 TB)), and other equipment (4 Dual Cell-based QS20 blades, a Voltaire 10 Gbps PCI express adapter, and 4 system x3755 login nodes (quad socket, dual core 2.6 GHz Optrons with 8 GB local RAM)), all connected by 10 Gbps Infiniband [6].

Our parallel programs were coded in C and utilized the LAM implementation of the MPI standard. Supported by the OSC and the University of Notre Dame, it is used in our programs for message passing and associated operations during processing.

3.2. DCT Computation using the gg90 Algorithm

The first algorithm employed in this work, denoted gg90, is based on the modified DCT algorithm of [7]. It involves three basic steps:

1. First reorder the input data points by reversing the values with odd indices and interleaving them. For example, let $n = 8$ and assume $x[0..7]$ is the input vector. To create the reordered vector $x'[]$, let $x'[i] = x[i]$ for $i = 0, 2, 4, \text{ and } 6$, while $x'[i] = x[8 -$

- $i]$ for $i = 1, 3, 5,$ and 7 . In the end $x'[] = [x[0], x[7], x[2], x[5], x[4], x[3], x[6], x[1]]$.
- Next, calculate cosine and sine values for appropriate pairs of points and angles using the $gg90$ formula illustrated in Figure 1 below.

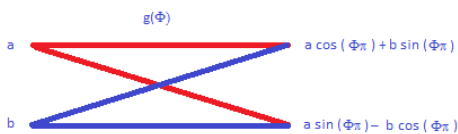


Figure 1: The $gg90$ Function.

- Finally, calculate the sum-differences of the results, as described by Figure 2 below.

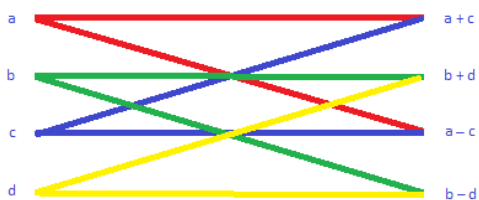


Figure 2: The $sumdiff$ Function.

These steps are repeated more-or-less recursively depending on the position of the data points. In the end, when the data point size is two, there is a last computation step to perform. In this step, a sum-difference of the two points, followed by division by $\sqrt{2}$, is performed. This step is pictured in Figure 3 below.

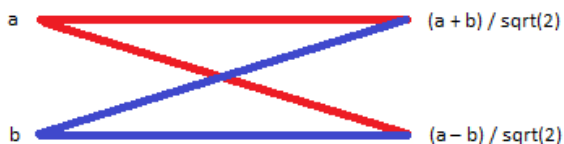


Figure 3: The final step in DCT.

The complete execution for an 8-point vector following reordering is shown in Figure 4 below. As you can see, after the first pass of the above steps, the problem is partitioned into top and bottom halves, with the $gg90$ function not applied to the top half. At the end, with only two data points remaining, the modified sum-difference operation is carried out to derive the final answer.

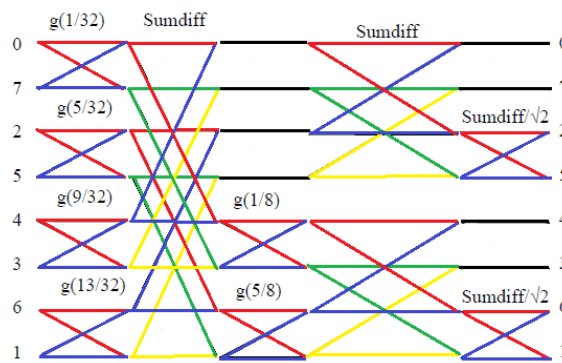


Figure 4: DCT-IV for eight data points.

3.3. DCT Computation via Lifting

Alternately, we can replace the calls to $gg90$ in Figure 4 with the lifting step pictured in Figure 5 below. Begin by pre-computing the R -value based on the sine and cosine. When this is known, we can compute $L1$, $L2$ and finally the upper output value, in that order. The idea is that one multiplication is eliminated by completing the calculations on the angles ahead of time.

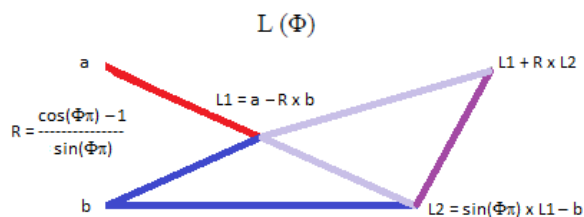


Figure 5: Lifting step for two data points.

3.4. Comparison

In order to determine our next step, we implemented the DCT using both $gg90$ and lifting, and ran the programs on one node of the UA cluster for varying numbers of points. Later, undergraduate students in our parallel processing course were given the code, instructed to modify it as they chose to improve execution times, and asked to repeat our experiment on a smaller data set. The results are shown in Tables 1 and 2 below. As can be seen, the lack of a multiplication operation does not seem to compensate for the added complexity of computation, with the gap growing steadily as points are added to the data set.

N	gg90	Lifting	$L(\Phi)/g(\Phi)$
128	124	151	1.22
256	215	296	1.38
512	437	649	1.49
1024	848	1557	1.84
2048	1658	4094	2.47
4096	3411	12,487	3.66
8192	7225	41,265	5.71
16384	15,066	148,190	9.84

Table 1: Timing (in μ secs) of gg90 and lifting algorithms on one processor, initial experiments.

N	gg90	Lifting	$L(\Phi)/g(\Phi)$
32	9	12	1.33
64	13	23	1.77
128	28	49	1.75
256	53	106	2.00
512	106	229	2.16

Table 2: Timing (in μ secs) of gg90 and lifting algorithms on one processor, student experiments.

The only remaining question regarding lifting is if the expense of the computation could be mitigated by the addition of processors. To investigate this possibility, we re-implemented the lifting code to execute in parallel using processors linked via MPI. The results appear in Table 3 below. As seen, the increased communication overhead overwhelms any potential gain in efficiency. In the end, we were forced to conclude that lifting would not be useful in future experiments involving our current equipment.

N	Initial experiments			Student exps.	
	1 proc	2 procs	4 procs	1 proc	2 procs
128	158	2530	3033	70	2904
256	300	6643	7090	148	7160
512	679	16,555	16,740	383	17,944
1024	1516	40,897	41,399		
2048	4010	129,486	129,200		
4096	12,242	442,862	441,023		
8192	40,544	1,436,512	1,439,811		

Table 3: Timing (in μ secs) of lifting algorithm on 1, 2 and 4 processors.

3.5. Experimental Results with the FFT

As a consequence of these first trials, we chose to proceed focusing exclusively on DCTs incorporating

the gg90 step. We also considered previously published results from a similar project in the University of Akron Biomedical Engineering program concerning MRI imaging [8] for comparison.

Three different trials were conducted. First, FFTs involving only real numbers were computed with a single processor. Next, two processors were employed. One processor received the real components of the input data points, the other the imaginary parts. The processors work independently on their computations, then exchange results when finished. One of the two processors recombines real and imaginary data and reports the final results.

The most complicated of the experiments involved the use of six processors. The basic workflow is described by Figure 6 below. Two processors get real and complex data, respectively, as before. Each of these processors then assigns work to two other processors, which compute the DCTs and return the results. When all processing is complete, one master processor sends its results to the other (in this case, the real data is transmitted to the processor holding the imaginary data), who then combines the two sets of figures into the final results as before.

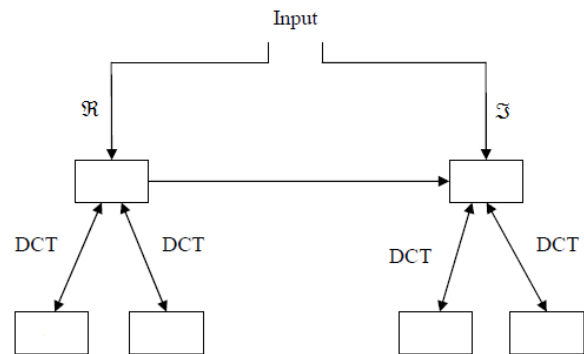


Figure 6: Implementation of FFT on six processors.

Our results are summarized in Table 4 below. Both the “UA” and “Biomed” columns in the table represent timings obtained on the University of Akron Computer Science cluster described above, while numbers in the “OSC” column were derived on their cluster.

N	Real FFT, 1 proc		
	UA	OSC	Biomed
128	113	123	
256	202	233	68
512	431	459	204
1024	799	792	486
2048	1621	1563	1170
4096	3472	3008	2800
8192	7018	5961	7000
16384	15,163	12,846	18,100
32768	31,153	27,781	46,900

N	Complex FFT, 2 procs		
	UA	OSC	Biomed
128	260	151	
256	415	269	66
512	721	519	199
1024	1266	929	456
2048	2401	1704	1040
4096	4284	3472	2330
8192	8458	6582	7000
16384	17,256	13,775	17,900
32768	35,675	30,410	47,000

N	Complex FFT, 6 procs	
	UA	OSC
128		529
256	1923	647
512	2335	838
1024	4066	1097
2048	7789	1832
4096	13,401	3851
8192	27,205	6609
16384	51,167	12,737
32768	120,313	26,613

Table 4: Timing (in μ secs) of experiments on FFTs.

The results are promising but reveal how much remains to be done. The faster hardware at the OSC does not show an effect on the 1-processor real case until N exceeds 1000 points. The improved communications backbone in the Glenn cluster makes a dramatic difference in the parallel complex trials when compared to the corresponding experiment on the Akron cluster, but not enough to justify the approach yet. On the other hand, we see that the use of the gg90 algorithm in the calculation of FFTs leads to improved results when compared to the Biomed experiments for large values of N . Similarly, the improved execution times on the OSC

cluster for the 6-processor tests over the 2-processor tests for large N indicate that there is a gain here if the communication costs can be sufficiently offset and that we are on the right track.

4. Conclusions and Future Work

These initial results indicate that there is hope for this direction but a lot of work must be completed. Better designed parallel implementations are part of the answer. So may be computing platforms with smaller communication penalties, such as GPU computing (i.e. NVIDIA[®] chips and CUDA[™] programming). Finally, we are far enough into the patent process now that we can talk more publically about Interleaved FFTs and hope to explore their application to specific problems, like MRIs.

5. Acknowledgements

The authors thank Profs. Stu Clary, Kathy Liszka and Wolfgang Pelz from the University of Akron faculty for their input into this work, as well as the scientists at the Ohio Supercomputer Center (OSC) in Columbus. We are also grateful to the students in the Spring 2009 section of Introduction to Parallel Processing at the University of Akron for their help in double-checking experimental results, particularly Rob Lipstreu, Blake Miner, Kyle Patterson and Jing Yu.

6. References

1. Wilkinson, Barry and Allen, Michael. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson Prentice Hall, 2005.
2. Quinn, Michael J. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.
3. Mugler, Dale H. The Centered Discrete Fourier Transform and a Parallel Implementation of the FFT. In *Proc. of the IEEE International Conference on Acoustics, Speech and Signal Processing*, to appear, 2011.

4. Mugler, Dale H. *The New Interleaved Fast Fourier Transform*. WIPO Patent Application WO/2007/100666, pending.
5. Wikipedia. Fourier transform. [Online] March 2011. http://en.wikipedia.org/wiki/Fourier_transform.
6. Rao, K.R. and Yip, P. *Discrete Cosine Transform: Algorithms, Advantages, Applications*. Academic Press, Inc., 1990.
7. Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L. and Stein, Clifford. *Introduction to Algorithms*. The MIT Press, 2009.
8. Ohio Supercomputer Center. High Performance Computing Systems. [Online] December 2008. <http://www.osc.edu/supercomputing/hardware/>.
9. Huang, H., Rahardia, S., Yu, R. and Lin, X. Integer MDCT with Enhanced Approximation of the DCT-IV. *IEEE Transactions on Signal Processing*, 54: 1156 - 1159, 2006.
10. Misal, Nilimb. *A Fast Parallel Method of Interleaved FFT for Magnetic Resonance Imaging*. M.S. Thesis, Dept. of Biomedical Engineering, The University of Akron, 2005.

Parallel Processing of Geospatial Time-series Data

Monte Lunacek, Peter Graf, and Wesley Jones

National Renewable Energy Laboratory, Golden CO 80401

Abstract—*One way of quantifying how much sunlight the earth receives, at a particular location and time, is to compute a Cloud Index from satellite images. This calculation involves processing large files of data that are collected in a way that makes time-series analysis extremely time consuming on a single computer. There are several ways to render this memory-intensive problem more feasible using parallel programming. Our implementation uses the Message Passing Interface (MPI) protocol and is an excellent balance between implementation complexity and execution time. We believe that this will also scale well as more complex algorithms for computing the Cloud Index emerge. This work extends to any domain where time-series analysis is needed on large data sets that are collected as a function of time.*

Keywords: Parallel programming, Data analysis, Memory management

1. Introduction

Collecting geographic data as a function of time is extremely common in climate and weather modeling. This creates a storage environment where the data necessary for time-series analysis is likely spread over many, often very large, files. We have recently been working on an application with this general characteristic. Our implementation estimates amount of sunlight that reaches the surface of the earth at different locations and times during a year. This provides critical information for making informed decisions about how well a particular region can harness solar technology. Unfortunately, there is a component of this process that cannot be computed in a reasonable amount of time because it requires simultaneous access to many large files that cannot all reside in memory.

In this paper, we use the Message Passing Interface (MPI) [5] protocol to communicate and distribute a fraction of the large data set to other processors such that each processor has the computational data in memory necessary for its part of the analysis. This allows us to perform our calculations in parallel on a process that would be extremely impractical to complete on a single computer. This approach does not make any assumptions about the type of data storage or how it can be accessed. We compare and discuss when this type of method may be preferred to using a parallel input and output scheme.

Our application illustrates a less common use of parallel computing. It is an excellent example of how parallel computing can render a memory-intensive process, one that

might not be possible, or at the very least, require an extremely careful and time-consuming serial implementation, feasible in a reasonable amount of time. Often High Performance Computing (HPC) is discussed in the context of CPU cycles; systems that are computationally intensive can distribute work across many cores and the resulting task will hopefully complete much faster when run in parallel. The model we describe is not computationally intensive at all, but limited by the amount of memory needed to ensure that all the components necessary for the calculation are available.

The process of more accurately modeling the amount of sunlight that reaches earth's surface builds on existing knowledge. Geostationary satellites, such as the Geostationary Operational Environmental Satellites, or GOES (first introduced in 1984 [4]), provide spatial and temporal spectroscopic information that we are using to compute a Cloud Index (CI), which measures a cloud's impact on light. Perez *et al.*[6] outline a method where this information, derived from satellite images, is combined with the Clear SKy Model [2], [1]—an approximation that assumes there are no clouds—to provide an improved picture of the amount of sunlight that reaches the surface of the earth at different times during the day, and different locations across the United States. This more precise model is called the All Sky Model (ASM).

The cloud index calculation is the component of the overall process that stands out because, unlike other stages in the workflow, the CI calculation requires temporal data that is sparse in memory due, in part, to the way data is collected and organized. This creates a bottleneck in the All Sky Model calculation because there is not enough memory on any single computer to calculate the Cloud Index in an efficient way.

In this paper, we describe the workflow that takes satellite images and transforms them into an All Sky Model. We give a detailed description of the Cloud Index calculation in order to discuss the various ways that this part of the process can be computed in a practical amount of time. We use this as a means for describing and evaluating solutions to memory-intensive computing problems. Finally we describe the results we generated using MPI communication and discuss why using MPI communication strikes a reasonable balance between implementation complexity and execution time.

2. All Sky Model

The GOES satellite is a geostationary satellite. It orbits the earth at the same speed as the earth rotates, which keeps its perspective on the earth the same at all times. Every day, GOES captures 83 images of the earth taken at different Universal Coordinated Times (UTC). The *full disk image* covers a projection of the entire earth. The *extended northern hemisphere* is a sub-image that is focused on North America and the northern half of South America. Each image is stored in the GOES Variable Format (GVAR) and contains the spectroscopic information used to compute the cloud index. Figure 1 shows the full image taken by the GOES satellite as well as the sub-image that represents the extended northern hemisphere section.

The raw images transmitted by the GOES satellite are converted to McIDAS AREA files [7] and indexed in a way that is tied to latitude and longitude for each pixel in the file. During this phase, the original raw image is condensed from a 1km resolution to a 5km resolution. For the extended northern hemisphere, this creates a file that has 1702×2002 values for each UTC time period of each day. Future implementations will work on files that have a different degree of resolution, including the maximum resolution of 1km, which will increase the size of the McIDAS AREA data structure by a factor of 25.

There is a one-to-one relationship between the indices of the McIDAS AREA file and the latitude and longitude on the earth. For example, Boulder, Colorado is located at index location (495, 497) in the AREA file.

The conversion from GVAR to AREA is necessary in part because the image values need to be corrected and normalized such that their brightness is comparable at different times during the day. At noon, the sun is at 90 degrees, whereas just before sunset, it is low on the horizon and has an angle close to zero degrees. A normalization of the values makes comparing the brightness at different times of the day possible. The pixels of the resulting normalized images are called Digital Number (DN) values. Unlike the GVAR and AREA files, these values are grouped by day. Since each day contains 83 UTC slices, the DN files contain $83 \times 1702 \times 2002$ floating point values. When implemented using a 4 byte floating point number, the entire file is roughly 1.05 Giga bytes in size ($4 \cdot 83 \cdot 1702 \cdot 2002$).

This large file is used to create the Cloud Index (CI), which is also stored by day in the same format and dimensions. A *Clear Sky Model* is used to generate surface irradiance when there are no clouds. This is combined with the Cloud Index to produce the final *All Sky Model* index. This metric quantifies how much sunlight hits the surface of the earth at each point in time. Figure 2 shows a diagram of the data process from GVAR to AREA to DN. The CI and ASM produce files in the same format as DN.

For the remainder of the paper, the x -dimension refers to the UTC value (time), the y -dimension refers to the latitude,

and the z refers to the longitude. Each day for a DN, CI, or ASM will have dimension (83, 1702, 2002).

2.1 The Cloud Index Calculation

The cloud index calculation requires DN values over a range of days. This reliance on neighboring DN values renders a serial solution impractical because, although a single day of DN values could fit into memory quite easily (≈ 1 Giga byte), most computers cannot load and access 30 to 60 days worth of DN values efficiently.

Part of the problem is data structure necessary for storing the DN values we need. In each file containing a single day, the current cloud index calculation only needs to access a small fraction of the data. Specifically, only a single pixel from each day is needed to compute the cloud index of that pixel for a given range of days. The cloud index calculation for a specific UTC, longitude, and latitude is completely independent of any other value in the file. This means that the DN values that we need to combine are very sparse within our data.

More complex calculations of the cloud index in the future may have a higher dependency on the information from each file, including values that have a similar location (e.g. latitude and longitude) and time (e.g. UTC). Yet this information is still sparse when considering a stream of data that contains 30 to 60 days. Furthermore, as the analysis of the CI data becomes more sophisticated, the calculation will necessarily become more complex and computationally expensive, even without using information from more than a single location and UTC. These future developments do not constitute a significant barrier to our implementation, which we describe below.

The current cloud index calculation is not computationally expensive. It is a simple averaging scheme that operates over a range of days, which we refer to as the *window*. The size of the window, which we call the *window-size*, is simply the number of days and represents the minimum number of DN values required to compute the cloud index ($\text{window-size} \cdot 1$). That is, we only need a single UTC, latitude, and longitude from each day in order to compute the cloud index for a give pixel.

This is really an instance of a more common type of problem. Anytime a large amount of data is collected as a function of time, it is possible that the data needed for analysis is spread in a sparse way over many large files. This becomes problematic when the number of files needed to perform an analysis becomes prohibitively large. In the next section, we discuss some ways of addressing this problem.

3. Implementation Strategies

As mentioned, it is not possible to load and access a complete window of days into memory. This means that regardless of how we implement the calculation, we must operate on a fraction of each day. We call this sub-component

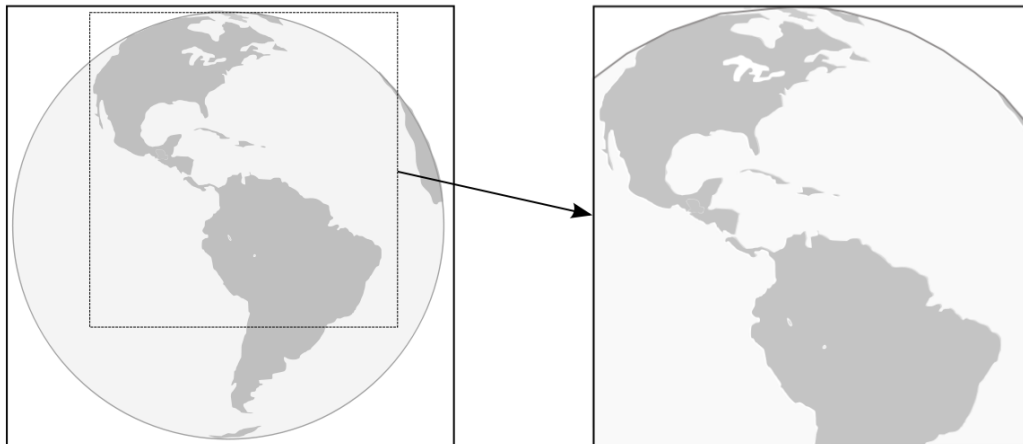


Fig. 1: The GOES satellite captures a full image of the earth (left) as well as a sub-section of the northern hemisphere. There are 83 images taken each day.

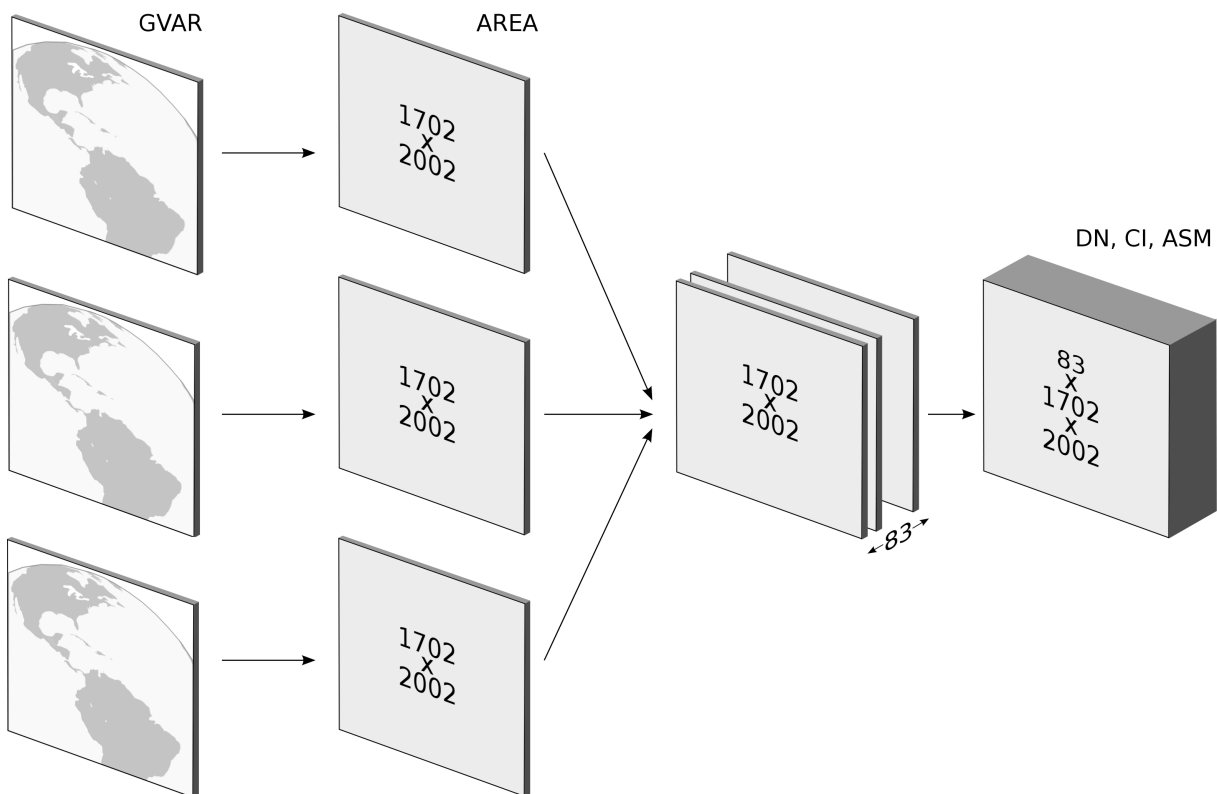


Fig. 2: An overview of the data flow calculating the All Sky Model. The initial images are in the GVAR format (left). These images are converted to McIDAS AREA files that link each pixel to an index that represents a longitude and latitude value. They are 1702×2002 in size. Finally, when creating the normalized DN values, the data is grouped in files by day, where each day contains 83 UTC images. The remainder of the workflow operates on files grouped by day. This includes the Cloud Index and the All Sky Model.

a *slice*. In our current implementation, a slice can be any subset of DN values because the Cloud Index calculation does not have a dependency on values within a day. However, future implementations may require a specific contiguous region. For example, a slice might be defined as the space inside the range of 0, 0, 0 to 83, 100, 100, which contains all the UTC values for the first 100 longitude and 100 latitude values in each day.

3.1 Serial Implementation

A serial implementation would, for example, be forced to read each file and keep only a fraction of the data such that the window-size \times slice is less than the total amount of memory available. This would be incredibly inefficient because all the data for each day would be read multiple times in order to process all the *slices* within the day.

With some file formats, a program can read only a portion of the stored data, which is often more efficient than reading the entire file. This is referred to as *partial IO*, and if available, would greatly improve the efficiency of the serial implementation by allowing each Cloud Index calculation to only read the slices that it needs. This would still require looping over the same time periods until an entire day is processed, but the redundancy of reading the entire file would be eliminated. HDF5 [3], for example, supports partial IO by allowing a program to select a subset of the file, which they call a *hyperslab*. The efficiency of partial IO in this case can vary depending on how you define the hyperslab with respect to the data organization.

The data structure we are using for the AREA files is certainly part of what makes a serial implementation difficult. For example, if the AREA files were not grouped into days, the bottleneck we are facing may be less prohibitive. However, the optimal data structure during each phase of the process is different and it is often not possible to define one way of storing data that is optimal for parallel processing at every stage of the workflow.

3.2 Parallel Implementation

The memory requirements of our problem limit the type of parallel implementations we can consider. For example, a shared memory approach is really not feasible because of the scale of our memory requirements.

Given the independent nature of each point in the file and the inherent limitation on memory, this problem is an excellent candidate for a distributed memory parallel implementation. While there are myriad ways this could be done, we discuss two options: 1) using parallel IO and 2) using MPI communication. In either case, the end goal is to have the necessary slices of data available to each processor and perform the calculation in parallel.

3.2.1 Parallel IO

It is possible for each processor to read its own slice of each day in parallel. Once a full window is available for processing, each processor computes the Cloud Index and writes the results in parallel to an output file. This strategy does not require large amounts of data to be passed between the processors using MPI communication, however, it does require the information to be passed on the filesystem.

There are some disadvantages to using parallel IO. Arguably, parallel IO is complex to implement and debug. It must also be supported by the file format and the file system. On a more subtle level, the way we access slices in a parallel IO context strongly depends on how the data is organized and stored, whereas in our implementation, the organization of the information in the file is completely decoupled from how we use that information. Therefore, there is no performance penalty for using non-contiguous sections of the file.

3.2.2 MPI Communication

In order to avoid the complexities and dependencies incurred when using parallel IO, we instead implemented a simple communication scheme where the master processor is the only rank that reads and writes, and each file is read only once. The master rank divides the file into slices and sends each working processor a portion of the overall file. When a rank has received window-size slices, it computes the cloud index and then discards the oldest day in order to make room for another slice. In this way, each rank contains a moving set of slices that are necessary for each cloud index calculation.

Figure 3 shows an example. The working processor collects a slice from each day until it has window-size slices. At this point, the working processor computes the cloud index. Then it discards the oldest day, which is no longer needed, and creates room for the next slice sent from the master.

The master processor only reads, writes, and communicates. When a processor sends back its slice containing the cloud index values, the master overwrites the DN values in memory as a means of conserving space. Once all the slices for a particular day are returned by the working processors, the master rank writes the memory to the cloud index file for that day. Figure 4 graphically explains the master's role. The master's primary responsibility is to read the file and send slices to each worker. Then it waits for the slices, that now contain the Cloud Index values, to return from each worker. This information is written to disk as the cloud index.

There are some limitations with this method. In order to process an entire day's data, there must be enough processors such that each working core can hold its share of the file and still have the full window-size in memory. If we continue to work off the assumption that memory will hold \approx one day, then this method requires at least window-size+1 processors.

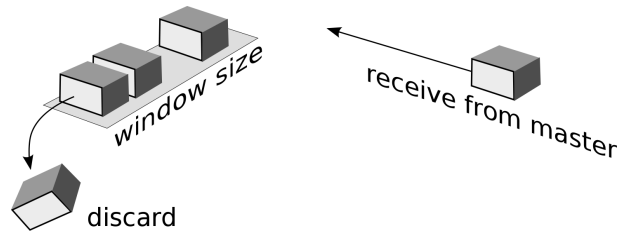


Fig. 3: The tasks performed by each working processor. The Cloud Index is computed once there are window-size slices in memory. Then the oldest day is discarded, making room for the next slice sent from the master.

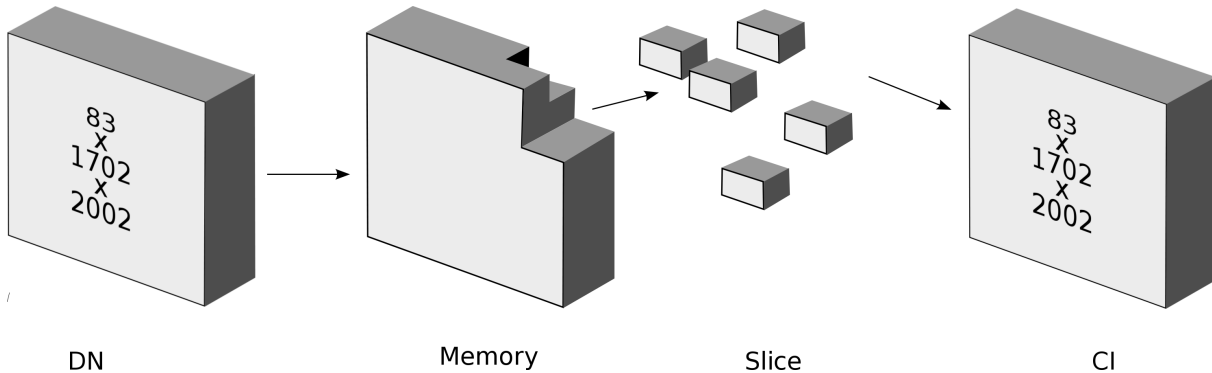


Fig. 4: The master processor breaks each day into slices that are sent to the working processors. Each worker sends back the CI result, which the master then writes to disk.

4. Empirical results and Discussion

The previous section discussed three possible implementations, and as mentioned, there are many other variations possible. In this section, we take measurements of our implementation and extrapolate these values to illustrate that using MPI communication to slice and distribute the data yields a large speedup when compared to a serial implementation, and that the additional benefit of using parallel IO is relatively small in comparison. The exact time values are not critical for the discussion.

We computed the cloud index over a one year period with a window size of 28 days using a total of 64 processors in about 145 minutes. The master divided each file into 63 slices and sent these to the workers. We found that it took approximately 4 seconds for the master processor to read and write each file. It took an additional 4 seconds for the master to send each worker its slice, and about the same amount of time to receive the results. Some processors begin their calculation while the master is still sending data. Similarly, the master processor received data from the processors that finish early while some workers are still processing. The computation time for each cloud index day was about 12 seconds, with a master wait time of approximately 8 seconds. In total, a one year calculation took 145 minutes (e.g. $(4 + 4 + 8 + 4 + 4) \cdot 365/60$).

A serial implementation would want to maximize its

memory usage. For the sake of argument, let's assume that the slice size was double. This means that a serial implementation would need to loop over each day 32 times instead of 64. This also doubles the amount of time the Cloud Index calculation takes (because the slices are twice as big). We estimate that processing each day would take about 4 seconds to read the file, 24 seconds to process the file, and another 4 seconds to write the file. This would need to happen 32 times for a total of $32(4 + 24 + 4) = 1024$ seconds. One year on a single computer would take about $(1024)/60 = 6,230$ minutes, or ≈ 4.3 days. Our parallel implementation has a speedup of about 40 times.

Using parallel IO on 64 processors, each day would take about 12 seconds to compute the cloud index, but it's possible that the parallel read and write time would take a fraction of what it takes to read the entire file. The best performance of the type of implementation would be $365 \cdot 12 = 4380$ seconds, or 73 minutes. This is about half of the time it takes using our implementation, but makes assumptions about how the data is stored on the underlying file system.

As mentioned earlier, we may want to include values that have a similar location and time in the Cloud Index calculation. Our implementation has an advantage here because the master has an entire day in memory and can, therefore, slice the data in any way with very little impact on performance. However, if a particular slice is desired using

parallel (or partial) IO, the performance is closely tied to how the data is organized and stored. HDF5, for example, calls this chunking. This limits the type of slices that can be processed efficiently.

5. Conclusions

Using MPI communication, we are able to compute the Cloud Index for the All Sky Model as a function of time. This computation would be extremely impractical without the use of parallel computing. We estimate that our implementation is approximately 40 times faster using 64 processors than a serial implementation would be. In a sense, using MPI communication is the middle road between the long wait necessary in a serial run, and the complex implementation details of using parallel IO. This trade-off strikes the right cord for our needs, but there are other ways we could speed up this calculation if needed.

Future work may consider both 1) arbitrary sections of a non-contiguous data, and 2) a more complex CI calculation. There are two important points here. First, this memory-intensive process could also become computationally more expensive in the future. If this happens, adding additional processors will help our implementation scale. Second, using MPI to slice and distribute the data will be more agile as different sub-sections of the data (e.g. slices of arbitrary latitude, longitude, and UTC values) are required because there does not exist a performance dependency on how the data is organized and stored. We conclude that our implementation scales well in this way.

Our discussion of the different implementations has some uncertainty. Unfortunately, there is very little discussion in the literature that includes actual results for problems of this type. Future work may implement different procedures in order to shed light on solutions to problems of this type.

At a higher level, our application belongs to a class of problems where time-series analysis must be performed on data sets that are collected and stored in a way that may render analysis impractical on a single computer. Continuing to explore ways to manage this type of data, and continuing to understand the trade-offs, can lead to more widespread disseminations of the techniques we use to understand our data.

Acknowledgements

This work was supported by the U.S. Department of Energy under Contract No. DE-AC36-08-GO28308 with the National Renewable Energy Laboratory.

References

- [1] P. Ineichen and R. Perez. A new airmass independent formulation for the linke turbidity coefficient. *Solar Energy*, 73, 2002.
- [2] F. Kasten. Parametrisierung der globalstrahlung durch bedeckungsgrad und trubungsfaktor. *Annalen der Meteorologie Neue Folge*, 20, 1984.
- [3] Q. Koziol and R. Matzke. HDF5 Ð a new generation of HDF: Reference manual and user guide. *National Center for Supercomputing Applications, Champaign, Illinois, USA*, <http://hdf.ncsa.uiuc.edu/nra/HDF5>, 1998.
- [4] P. Menzel and J. Purdom. Introducing GOES-I: The first of a new generation of geostationary operational environmental satellites. *Bulletin of the American Meteorological Society*, 75, 1985.
- [5] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.
- [6] R. Perez, P. Ineichen, K. Moore, M. Kmiecik, C. Chain, R. George, and F. Vignola. A new operational model for satellite-derived irradiances: description and validation. *Solar Energy*, 2002.
- [7] E. A. Smith. The McIDAS system. *IEEE Transactions on Geoscience Electronics*, 1975.

A Parallel GPU Version of the Traveling Salesman Problem

Molly A. O'Neil, Dan Tamir, and Martin Burtscher

Department of Computer Science, Texas State University, San Marcos, TX

Abstract - *This paper describes and evaluates an implementation of iterative hill climbing with random restart for determining high-quality solutions to the traveling salesman problem. With 100,000 restarts, this algorithm finds the optimal solution for four out of five 100-city TSPLIB inputs and yields a tour that is only 0.07% longer than the optimum on the fifth input. The presented implementation is highly parallel and optimized for GPU-based execution. Running on a single GPU, it evaluates over 20 billion tour modifications per second. It takes 32 CPUs with 8 cores each (256 cores total) to match this performance.*

Keywords: Traveling Salesman Problem, Iterative Hill Climbing, GPGPU, Program Parallelization

1 Introduction

The traveling salesman problem (TSP) is one of the most commonly explored combinatorial optimization problems (COPs), often used as an early exploration ground for new approaches to COPs [1]. Consider a complete, undirected, weighted graph $G(V, E, W)$, where V is a set of vertices, E is a set of edges, and W is a set of edge weights. A Hamiltonian tour in G is a cycle that starts from a vertex $v_0 \in V$ and traverses all other vertices of G exactly once [1]. The symmetric TSP is a special case of the problem of finding a minimal Hamiltonian tour in a complete, undirected, planar, Euclidean, weighted graph in which the vertices represent cities, the edge weights represent the distances between the cities, and the distance from city v_A to city v_B is the same as the distance from city v_B to city v_A . The optimal TSP solution consists of the Hamiltonian tour that yields the minimum distance traveled.

Finding an optimal solution to TSP is NP-hard [2], so it is frequently approached using heuristic algorithms that find near-optimal tours. Constructive multi-start search algorithms, such as iterative hill climbing (IHC), are often applied to combinatorial optimization problems like TSP. These algorithms generate an initial solution and then attempt to improve it using heuristic techniques until a locally optimal solution, i.e., one that cannot be further improved, is reached. In each IHC step, a set of tour modifications, called *moves*, are evaluated to determine the best move [3], [4]. For instance, the tour can be adjusted by a heuristic such as 2-opt, which removes the edges (v_A, v_B) and (v_C, v_D) and adds edges (v_A, v_C) and (v_B, v_D) [1]. The IHC algorithm repeatedly

chooses the best move as the next step, reducing the length of the tour until it finds a locally optimal solution, then restarts with a new initial construction. This process of local improvements and restarts continues until the solution is sufficiently good or a limit on computing resources is reached [5]. IHC is used for several problems, including finding the maximal parsimony (phylogenetics) tree (MPT), where thousands if not millions of restarts are needed to find a good solution with high probability, making this approach computationally expensive. In this paper, TSP serves as a test bed for improving IHC implementations for solving problems such as MPT.

The past decade has seen a rise in the use of graphics processing units (GPUs) as general-purpose computing devices that can efficiently accelerate many non-graphics programs, especially vector- and matrix-based codes exhibiting a lot of parallelism with low synchronization requirements. Because their hardware is primarily designed to perform complex computations on blocks of pixels at high speed and with wide parallelism, GPU architectures differ substantially from conventional CPU hardware. This can make it difficult to write efficient implementations of non-graphics algorithms for GPUs.

For example, NVIDIA GPUs require sets of 32 program threads, called warps, to execute the same instruction in every clock cycle or wait. When not all threads in a warp can execute the same instruction, the warp is subdivided by the hardware into sets of threads such that all threads in a set execute the same instruction. These sets execute serially until they re-converge, resulting in a loss of parallelism.

The memory subsystem is also optimized for warp-based processing. If a warp accesses 32 consecutive words in memory, the hardware merges the 32 reads or writes into one coalesced memory access that is as fast as a single non-coalesced access, subject to alignment and word-size constraints. Thus, it is crucial to use coalesced memory accesses to exploit the GPU's high memory bandwidth.

The 32 processing elements (PEs) within each streaming multiprocessor (SM) of a GPU share a pool of threads called a thread block, synchronization hardware, and a software-controlled cache called shared memory. A warp can simultaneously access up to 32 distinct words in shared memory as long as the words reside in different memory banks. Barrier synchronization between the threads in an SM takes one clock cycle if all threads reach the barrier together.

The PEs are fed with warps for execution in multi-threading style to hide latencies. Thus, it is paramount for

good performance to have many active resident warps in each SM. In other words, GPUs require thousands of simultaneously running threads, i.e., large amounts of parallelism to achieve maximum performance.

The SMs operate largely independently. They can only communicate with each other through global memory (DRAM). Thus, synchronization between SMs must be done using atomic operations on global memory locations, meaning that GPUs are most effective at accelerating codes with low sharing requirements.

The large amount of parallelism and wide memory buses make GPUs well suited to speed up codes displaying high computational intensity and little synchronization. For such codes, GPUs have demonstrated a substantial advantage over CPUs in terms of performance per dollar and performance per transistor [6] as well as performance per watt [7]. GPU implementations of these applications can be dozens of times faster than optimized parallel CPU implementations [8].

This paper explains how we parallelized and optimized the IHC algorithm for TSP so that it can reap the benefits of GPU acceleration. Our implementation running on one GPU chip is 62 times faster than the corresponding serial CPU code, 7.8 times faster than an 8-core Xeon CPU chip, and about as fast as 256 CPU cores (32 CPU chips) running an equally optimized pthreads implementation. For symmetric, planar, 100-city problems with 100,000 random restarts, our code finds the optimal solution for four out of five TSPLIB inputs and is 0.07% off on the fifth input. Our open-source CUDA implementation is freely available for download at http://www.cs.txstate.edu/~burtscher/research/TSP_GPU/.

2 Parallelization and optimization

This section explains how we implemented, optimized, and parallelized the IHC algorithm for the TSP problem. In this discussion, we assume symmetric 100-city problems with 100,000 random restarts.

2.1 Parallelization

There are several ways to parallelize this algorithm. The 100,000 climbers are independent and can be processed in any order, including concurrently. However, load balance is a potential problem when parallelizing the climbers as they require different numbers of IHC steps to reach a local optimum. Within a climber, each IHC step depends on the previous step and therefore has to execute serially. In our implementation, every IHC step evaluates 4851 opt-2 moves. These moves are independent and can be run in parallel, but they require a reduction operation at the end to determine the move that yields the largest reduction in tour length. This reduction can be performed in $\log_2(4851) \approx 13$ steps but necessitates synchronization and data exchange, which may be slow.

Because modern GPUs require tens of thousands of parallel threads that perform very similar tasks to unleash their full performance, we decided to run the independent climbers in parallel. This approach results not only in the highest degree of parallelism but also in the least amount of synchronization and data exchange. However, the climbers perform varying numbers of IHC steps to reach a local optimum. We measured between 84 and 124 steps with an average of 103.3. Since we launch 14,336 threads on the GPU, the average thread processes only 7 climbers, which results in load imbalance and consequently poor scaling. In contrast, we launch no more than 256 threads on the CPU, yielding an average of 391 climbers per thread, which is enough to average out the number of IHC steps performed by each thread. Thus, load balance is not an issue with the CPU code but is significant in the GPU code. Because load balancing imposes synchronization and serialization overheads, the pthreads code actually runs faster without load balancing whereas the CUDA code runs faster with load balancing. Hence, we ended up with the following implementations.

Our pthreads code statically assigns equal (± 1) numbers of climbers to each thread. The threads run independently to find the best solution among their climbers and only execute a single critical section at the end to determine the best solution among all threads. The GPU code, in contrast, only assigns a single initial climber to each thread. When a local minimum is reached, the thread checks whether this minimum is smaller than the currently best solution. If it is, the best solution is updated using an atomic compare-and-swap instruction. Then, the next climber is obtained from a global worklist using an atomic increment. Threads terminate when the worklist is empty.

2.2 Code optimization

Our serial, pthreads, and CUDA implementations use essentially identical code for evaluating the opt-2 moves, which takes the vast majority of the runtime. This code section comprises two nested `for` loops that iterate over the cities to form pairs of cities between which the tour is reversed. The CUDA code differs from the serial and pthreads code in that we manually moved two loop-invariant computations out of the inner loop and specified that the inner loop be unrolled eight times. This was not necessary in the serial and pthreads codes as the C compiler automatically performs these optimizations.

Because GPUs are only fast if sets of 32 threads, i.e., warps, perform the same work (on different data) at the same time, our implementation always considers 4851 city pairs in each IHC step. In particular, the outer `i`-loop iterates from the 1st to the 98th city while the inner `j`-loop iterates from the `i+2nd` to the 100th city. Note that this approach avoids duplications in city pairs due to symmetry as well as pairs of adjacent cities that never result in a change of the tour length. Note that we always compute the tour length for all 4851 city pairs, including the ones that did not change from

the previous IHC step, because re-computing them is faster than recording and retrieving this information.

We optimized the loop nest by saving values fetched from memory in register variables so that later iterations can quickly access them. For example, even though we need four city IDs (the i^{th} , $i+1^{\text{st}}$, j^{th} , and $j+1^{\text{st}}$) in every iteration, the inner loop body only fetches the $j+1^{\text{st}}$ city ID from memory as the remaining values have been fetched earlier and are “cached” in variables. Similarly, each opt-2 move needs four distance values from a two-dimensional matrix (i^{th} to $i+1^{\text{st}}$ city, j^{th} to $j+1^{\text{st}}$ city, i^{th} to j^{th} city, and $i+1^{\text{st}}$ to $j+1^{\text{st}}$ city, where the 101st city is the same as the 1st city). Nevertheless, the code only fully evaluates one distance, partially evaluates two of the distances (by accessing a vector, i.e., a predetermined row of the matrix), and uses a cached value for the fourth distance to minimize computations and memory accesses. Aside from these operations, the inner loop only contains assignment statements that copy one scalar variable into another and an `if` statement to check whether a new optimum has been found.

To further boost the performance, the loop nest never actually computes the tour cost. It only calculates how much shorter an opt-2 move makes the current tour and picks the move that results in the greatest savings. As long as an IHC step results in a reduction in tour cost, the corresponding best opt-2 move is applied, i.e., the selected tour segment is reversed, and the next IHC step is initiated. Only once a local optimum has been reached is the tour cost finally computed. If this cost is lower than the previously found shortest tour, the new tour is written back to global memory and the shortest tour is updated. Otherwise, the new tour is simply discarded to avoid unnecessary memory writes.

To make the results deterministic and to simplify verification, the random seed used for generating a tour is the tour number (0 to 99,999). This guarantees that the length of the shortest tour is always the same, no matter in which order the 100,000 tours are processed. Because a cyclic rotation of a tour does not yield a new tour, the first city, which is also the last city, can be fixed without loss of generality. This enables simplifying and accelerating the program by hard coding the ID of the first city. Our code contains several other minute enhancements.

The CUDA code further contains GPU-specific optimizations that do not apply to the CPU code. For instance, the two-dimensional distance matrix is allocated in shared memory, a software-controlled cache, so that accesses to it are always fast. The 1024 tours that are evaluated concurrently in an SM are too large to fit in shared memory. Thus, we allocate them in global memory (DRAM). To still be able to access them quickly, the code first copies the tours into local memory, which is part of the global memory but ensures that every tour access in the two nested loops is fully coalesced. Other GPU optimizations include limiting thread divergence to rarely executed code sections and minimizing CPU/GPU transfers to just 40 kB initially to copy the distance matrix to the GPU and 108 bytes in the end to

copy the best tour, its cost, and its tour number back to the CPU. Note that, other than generating the distance matrix and printing the result, our implementation runs the entire TSP algorithm on the GPU.

3 Related work

Most previous GPU-based approaches to the traveling salesman problem use the Max-Min Ant System (MMAS) algorithm [9]. This algorithm is a variant of Ant Colony Optimization (ACO), a metaheuristic algorithm based on the natural ability of ants to discover, collaboratively, the shortest path between their nest and a food source by depositing pheromone along their traveled paths. ACO algorithms simulate the behavior of individual ants, which construct tours around a graph based on the strength of evaporating pheromone trails left by other ants. Dorigo and Gambardella first presented this algorithm applied as a distributed TSP solver [10]. ACO algorithms spend the majority of their computation time in the tour construction phase [11], and because ants travel independently and each ant constructs a complete solution based only on the previous iterations’ pheromone matrix, this phase is highly parallelizable.

Bai et al. detail a CUDA implementation of the parallel MMAS algorithm in which multiple ant colonies are simulated concurrently on the GPU, one for each thread block, with the tours of individual ants within each colony also parallelized [12]. This implementation achieves up to a 32x speedup over a serial CPU version under the same workload, though without finding the optimal solution in some cases. Jiening et al. present a C++ and Cg implementation of the MMAS algorithm with up to a 1.4x speedup over the CPU implementation, which finds the optimal tour on a 30-city input [13]. You describes a CUDA implementation of a parallel ACO algorithm [14], with each thread on the GPU responsible for the travel of a single ant from a unique starting location, achieving up to a 20x speedup over a serial CPU implementation. Cecilia et al. present several GPU-based, data-parallel strategies for both the tour construction and pheromone update stages of the ACO algorithm, achieving a 28x speedup for the tour stage and a 20x speedup for the pheromone update stage over sequential CPU code [15]. Many of the prior works on GPU-based ant colony solvers compare solution quality only against a serial ACO implementation and do not address how often either implementation discovers the optimum TSP solution.

There are also heterogeneous implementations of ACO algorithms, which implement only part of the TSP solver on the GPU. Delévacq et al. implement a parallel approach to the MMAS algorithm that performs tour construction on the GPU and pheromone update on the CPU [11]. Next, they compare their implementation against their GPU version of the original ACO algorithm [16], achieving better solution quality (though still suboptimal in some cases) and up to a 3.6x speedup. Fu et al. describe an MMAS implementation in MATLAB with the tour construction performed on the

GPU and the updates performed on the CPU [17]. This implementation achieves roughly a 32x speedup over sequential CPU code, but with slightly lower solution quality compared to the CPU implementation.

There also exists a recent genetic algorithm-based TSP solver in CUDA, presented by Fujimoto and Tsutsui in 2011 [18]. This work parallelizes TSP using the genetic crossover operator and 2-opt local search. Their CUDA implementation on a GTX-285 is up to 24.2x faster than a single-core CPU version, allowing an error ratio over the optimal trip cost of up to 0.5%.

To the best of our knowledge, this paper presents the first GPU implementation of the IHC algorithm for solving the TSP problem. Our IHC approach may be better suited for GPUs than previously proposed algorithms as it yields larger speedups over both serial and parallel CPU implementations while, at the same time, achieving very high solution quality.

4 Evaluation methodology

We evaluated our GPU implementation of TSP on an NVIDIA Tesla C2050 graphics card, which has CUDA compute capability 2.0 [19]. This GPU is equipped with 14 streaming multiprocessors (SMs), each with 32 cores, for a total of 448 cores running at 1.15 GHz and sharing 3 GB of global memory. Each multiprocessor is configured with 48 kB of shared memory and a 16 kB L1 cache. All SMs share a 768 kB L2 cache. Each SM has 32,768 registers that are shared among the threads allocated to the multiprocessor. The CUDA code was compiled with *nvcc* version 3.2 using the “-O3 -arch=sm_20” flags.

We ran the pthreads and sequential CPU implementations on the Nautilus supercomputer at NICS, which contains 128 2.0 GHz 8-core Xeon X7550 CPUs sharing 4 TB

of main memory. The pthreads and sequential codes were compiled with *icc* version 11.1, with the “-O3 -xW -pthread” flags for the pthreads version and the “-O3 -xW” flags for the sequential version.

We instrumented the three implementations to measure the runtime of everything except the reading in of the 100 city coordinates and the generation of the distance matrix from these coordinates. We tested all implementations on five TSPLIB benchmarks containing 100 cities [20].

5 Results

Figure 1 plots the runtimes (in milliseconds) of our three IHC implementations on the kroE100 TSPLIB input, with the minimum, median, and maximum runtime of three runs plotted separately. The runtimes for other 100-city inputs and different random restarts are very similar. The median runtime is listed above the columns. The results show that our GPU implementation’s median runtime, at 2.497 seconds, is slightly under that of the parallel CPU version run with 256 threads and dramatically less than that produced by the sequential CPU code (2.58 minutes). Unlike the GPU version, which produces highly consistent runtimes, the pthreads runtime at higher thread counts varies substantially between executions. In fact, in some experiments, it already started varying with 16 threads, i.e., the problem seems to appear as soon as multiple CPU chips are used. Since we made sure that there is no false sharing and only a minimal amount of true sharing in our pthreads implementation, we assume the variability is caused by interference from other jobs that were running at the same time on this large shared memory machine.

Figure 2 displays the minimum, median, and maximum speedups of the pthreads and GPU implementations relative to the sequential CPU implementation. Again, we see that

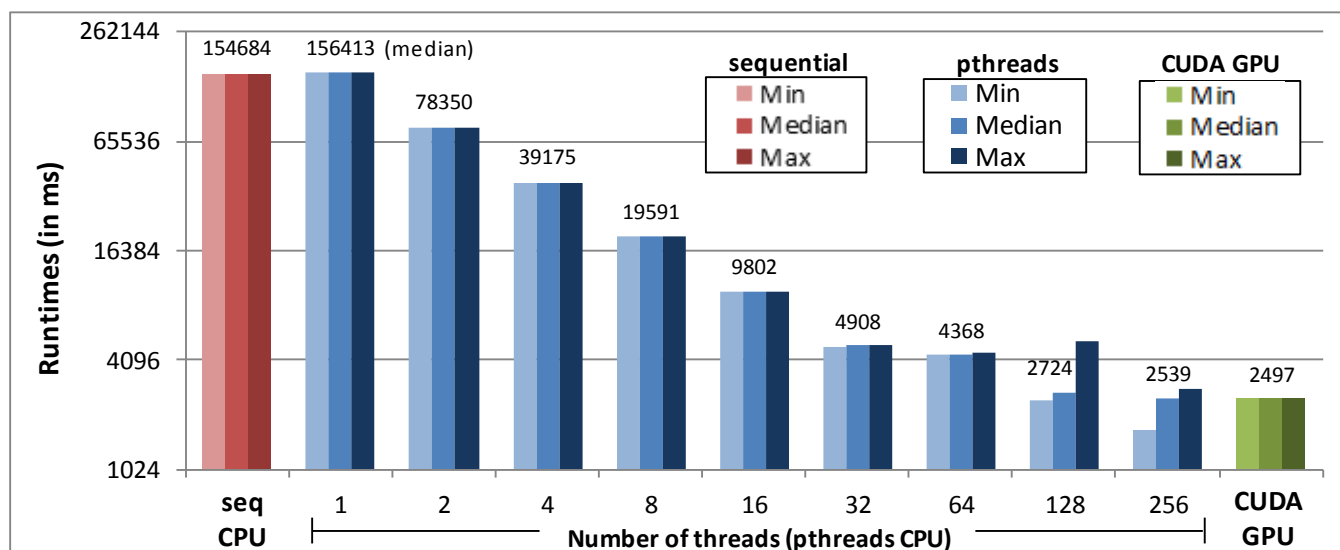


Figure 1. Minimum, median, and maximum runtimes (in milliseconds) of the three TSP implementations (note that this graph is logarithmic)

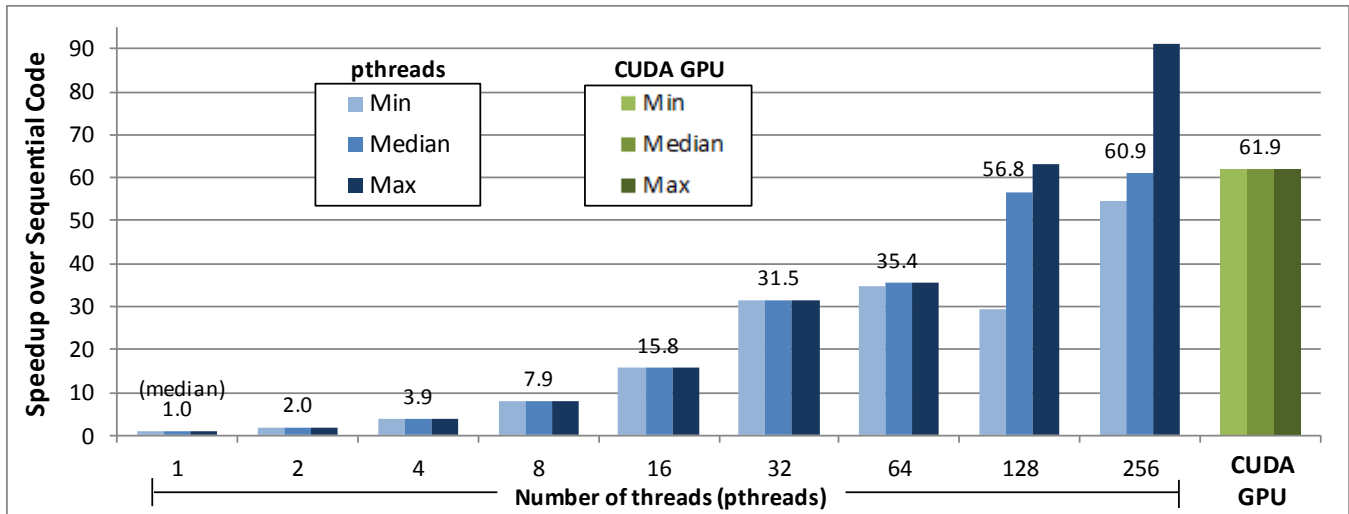


Figure 2. Minimum, median, and maximum speedup of the pthreads and GPU implementations relative to the serial CPU implementation

the GPU version produces consistent speedups whereas the pthreads version with 128 and 256 threads demonstrates significant performance variance. The pthreads code scales almost perfectly to 32 cores, indicating that it does not suffer from false sharing, load imbalance, serialization, or other parallelization overheads. However, scaling is poor beyond 32 threads, possibly due to the increasing thread startup cost. Additional experiments with different random restarts resulted in the same scaling trends. While the maximum speedup offered by the 256-thread CPU version exceeds that of the GPU implementation, the GPU code outperforms the pthreads code in terms of median speedup. It achieves a consistent speedup of around 61.9 compared to the 256-thread pthreads version's median speedup of 60.9. This means that the GPU is capable of slightly exceeding the performance of 256 x86 cores or 32 CPUs with eight cores each on the IHC TSP algorithm.

The Nautilus supercomputer on which we tested the pthreads implementation has 2.0 GHz CPU cores. The sequential and pthreads implementations would benefit from CPUs with faster clocks. However, we found the GPU implementation to still offer a 50x speedup over the sequential implementation executed on a 2.53 GHz Intel Xeon, suggesting that the GPU solution offers a large performance advantage over the CPU implementation even for the fastest currently available CPUs.

Table 1 addresses the solution quality and shows the cost and number of the shortest tour found by the GPU implementation for five 100-city inputs from the TSPLIB library when using 100,000 random restarts. The optimal tour cost and the runtime for each input are shown as well. Our GPU code finds the optimal tour in all but one case, on kroE100, where the tour is 0.07% longer. Doubling the number of climbers to 200,000 allows the GPU code to find the optimal tour in the last case as well.

Table 1. Solution quality achieved by the GPU implementation for five 100-city inputs from TSPLIB

TSPLIB Database		CUDA GPU Solution Quality		
Name	Optimal Cost	Min. Tour Cost	Min. Tour #	Runtime (s)
kroA100	21,282	21,282	33,188	2.540
kroB100	22,141	22,141	5,969	2.499
kroC100	20,749	20,749	23,092	2.543
kroD100	21,294	21,294	32,142	2.497
kroE100	22,068	22,084	16,941	2.499
		22,068	117,583	4.952

6 Summary and conclusions

This paper explains how we parallelized and optimized the IHC algorithm for solving the TSP problem on GPUs. The results demonstrate that our implementation not only yields a high solution quality but also runs very quickly. It processes over 20 billion 2-opt moves per second on a single GPU, which is 62 times faster than an x86 core and as fast as 32 CPUs with 8 cores running a pthreads version of the same algorithm. Based on these results, we believe our approach may be better suited for GPU-based acceleration than the related ant colony and genetic algorithm-based TSP solvers that are available for GPUs.

7 Acknowledgments

This research was supported by an allocation of advanced computing resources provided by the National Science Foundation. Some of the computations were performed on Nautilus at the National Institute for Computational Sciences [21]. We thank NVIDIA Corporation for donating the GPU that was used to develop,

tune, and measure the CUDA implementation of the algorithm presented in this paper. We further thank Intel Corporation for donating the server on which the serial and pthreads codes were developed.

8 References

- [1] Johnson, D. and McGeoch, L. "The Traveling Salesman Problem: A Case Study in Local Optimization." *Local Search in Combinatorial Optimization*, by E. Aarts and J. Lenstra (Eds.), pp. 215-310. London: John Wiley and Sons, 1997.
- [2] Garey, M.R. and Johnson, D.S. "Computers and Intractability: A Guide to the Theory of NP-Completeness." San Francisco: W.H. Freeman, 1979.
- [3] Ambite, J. and Knoblock, C. "Planning by Rewriting." *Journal of Artificial Intelligence Research*, pp. 207-261. 2001.
- [4] Pitsoulis, L.S. and Resende, M.G.C. "Greedy Randomized Adaptive Search Procedures." *Handbook of Applied Optimization*. Oxford University Press, pp. 168-183. 2001.
- [5] Rego, C. and Glover, F. "Local Search and Metaheuristics." *The Traveling Salesman Problem and its Variations*, by G. Gutin and A.P. Punnen (Eds.), pp. 309-368. Dordrecht: Kluwer Academic Publishers, 2002.
- [6] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., and Purcell, T.J., "A Survey of General-Purpose Computation on Graphics Hardware." *Computer Graphics Forum*, Vol. 26, pp. 80-113. 2007.
- [7] Huang, S., Xiao, S., and Feng, W. "On the Energy Efficiency of Graphics Processing Units for Scientific Computing." *International Symposium on Parallel Distributed Processing*, pp. 1-8. 2009.
- [8] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., and Skadron, K. "A Performance Study of General-Purpose Applications on Graphics Processors Using CUDA," *Journal of Parallel and Distributed Computing*, Vol. 68, No. 10, pp. 1370-1380. 2008.
- [9] Stutzle, T. and Hoos, H.H. "MAX-MIN Ant System." *Future Gen. Comput. Syst.*, vol. 16, no. 9, pp. 889-914. June 2000.
- [10] Dorigo, M. and Gambardella, L.M. "Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem." *IEEE Transactions on Evolutionary Computation*, Vol. 1, No. 1, pp. 53-66. April 1997.
- [11] Delévacq, A., Delisle, P., and Krajecki, M. "Max-min Ant System on Graphics Processing Units." *Third International Conference on Metaheuristics and Nature Inspired Computing*. October 2010.
- [12] Bai, H., Yang, D.O., Li, X., He, L., and Yu, H. "MAX-MIN Ant System on GPU with CUDA." *Fourth International Conference on Innovative Computing, Information and Control*, pp. 801-804. December 2009.
- [13] Jiening, W., Jiankang, D., and Chunfeng, Z. "Implementation of Ant Colony Algorithm Based on GPU." *Sixth International Conference on Computer Graphics, Imaging and Visualization*, pp. 50-53. August 2009.
- [14] You, Y.-S. "Parallel Ant System for Traveling Salesman Problem on GPUs." *Eleventh Annual Conference on Genetic and Evolutionary Computation*. July 2009.
- [15] Cecilia, J.M., Garcia, J.M., Ujaldon, M., Nisbet, A., and Amos, M. "Parallelization Strategies for Ant Colony Optimisation on GPUs." *14th International Workshop on Nature Inspired Distributed Computing*. May 2011.
- [16] Delévacq, A., Delisle, P., Gravel, M., and Krajecki, M. "Parallel Ant Colony Optimization on Graphics Processing Units." *Sixteenth International Conference on Parallel and Distributed Processing Techniques and Applications*. July 2010.
- [17] Fu, J., Lei, L., and Zhou, G. "A Parallel Ant Colony Optimization Algorithm with GPU-Acceleration Based on All-in-Roulette Selection." *Third International Workshop on Advanced Computational Intelligence*, pp. 260-264. August 2010.
- [18] Fujimoto, N. and Tsutsui, S. "A Highly-Parallel TSP Solver for a GPU Computing Platform." *Lecture Notes in Computer Science*, Vol. 6046, pp. 264-271. 2011.
- [19] "NVIDIA's Next Generation CUDA Compute Architecture: Fermi." Whitepaper, NVIDIA Corporation. 2009.
- [20] Reinelt, G. "TSPLIB—A Traveling Salesman Problem Library." *ORSA Journal on Computing*, Vol. 3, No. 4, pp. 376-384. Fall 1991.
- [21] National Institute for Computational Sciences, <http://www.nics.tennessee.edu/>. Last accessed March 8, 2011.

Genetic algorithm based on number of children and height task for multiprocessor task Scheduling

Marjan Abdeyazdan¹,Vahid Arjmand²,Amir masoud Rahmani³, Hamid Raeis ghanavati⁴

¹ Department of Computer Engineering, Mahshahr branch, Islamic Azad University, mahshahr, Iran.
Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran.

e-mail: abdeyazdan87@yahoo.com

²Department of Computer Engineering, Mahshahr branch, Islamic Azad University, mahshahr, Iran.

e-mail: vahid.arjmand@gmail.com

³Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran.

e-mail: rahmani@sr.iau.ac.ir

⁴Mahshahr, Iran.

e-mail: Hamid_raeis40@yahoo.com

Abstract- *Due to optimal use of processors as well as spending less time, the task scheduling in multiprocessor systems is of great importance. This is one of the NP_hard problems and achieving the optimal schedule or finding the minimum schedule length, using the dynamic algorithm and back-tracking programming, would be time-consuming. Therefore, heuristic methods like genetic algorithms are suitable methods to schedule tasks in a multiprocessor system. In this paper, a new genetic algorithm is presented whose priority of tasks' execution is based on the number of their children and then height task in per group with number of children equal. The results show that our developed algorithm finds the near-optimal schedule in a reasonable computation time, compared to other heuristics.*

Keywords: Multi processor, Genetic algorithm, Schedule, Task graph, Distribute system.

1 Introduction

A big program could not have been performed on a single processor in a reasonable time. Therefore, it has to be divided into several tasks and the schedule length should be minimized applying appropriate scheduling in a multiprocessor system.

For mathematical modeling of task scheduling problem, Direct Acyclic Graph (DAG) is used since

each task is represented by its corresponding node in this graph. Presence of an edge from task t_i to task t_j means that while task t_i is not finished, task t_j can not start execution. The objective of scheduling a task graph onto a multiprocessor system is to allocate n tasks to m processors, as the priority task relations are observed and the completing time of the final task is reduced to minimum. Simply, if two tasks are scheduled on two different processors, the communication cost would be zero.

Scheduling in a multiprocessor system is an NP_Hard problem [1]. In traditional and dynamic methods, obtaining the best schedule is too time-consuming and often random execution of tasks needs less time. Then, in heuristic methods the best schedule is not necessarily obtained in a reasonable time; however the obtained solution is close to the best one. Many heuristic methods have been studied such as: min-min, max-min, duplex, MCT (Minimum Completion Time), MET (Minimum Execution Time) [2], SA (Simulated Annealing) [3, 4], tabu search [5]. One of the best heuristic methods on task scheduling in multiprocessor systems is genetic algorithm [3, 6, 7, 8, 9, 11]. In this paper, a new genetic algorithm is introduced which executes tasks with respect to their priorities, based on the number of their children and then based on height task in per group with number of children equal. Section 3 a new

The paper is structured as followed. Section 2 presents priority-based task scheduling. In method is explained

that suggests prioritized tasks based on the number of their children and then based on height task in per group with number of children equal. Section 4 elaborates on simulation and its result and Section 5 summarizes the achievements.

2 Priority-based task scheduling

2.1 Merge tasks

When combining a node with one of its parents, p , the start time of the siblings of n may be increased. To resolve the difficulty, the parent node, p , could be apply condition merge. However, if there are three condition then merge two nodes parent and child.

- a. Per current node only have one parent
 - b. Current node is only child for that parent
 - c. Execute time per task for current node between different processors less than average communication value (edge between parent and child) current node and that parent.
- $$dif(n_i) = \max(W(n_i, p_j) - \min(w(n_i, p_m))) < C(n_k, n_i)$$

2.2 Schedule length

The goal in scheduling problem is to minimize the schedule length. The time that the final task is completed on a processor is called the finishing time of that processor. The maximum finishing time between m processors is called TFT (Total Finishing Time) of the schedule or schedule length. TFT is calculated by Equation (1).

$$TFT = \text{Max} \{ \text{Finishing Time of Processor}_j \}; \text{ for } 1 \leq j \leq m \tag{1}$$

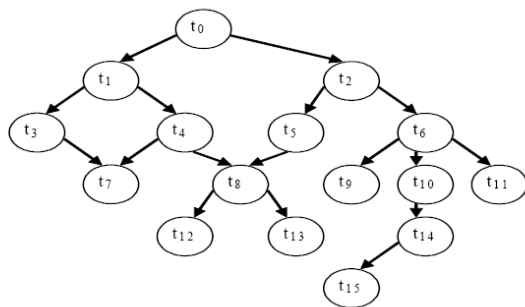


Figure 1: A task graph

Table 1: Hieght of tasks in Figure1

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
0	1	1	2	2	2	2	3	3	3	3	3	4	4	4	5

Table 2: Execution time of tasks

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
3	2	4	1	10	3	6	9	7	11	5	5	8	10	15	2

2.3 Prioritizing tasks based on the number of their children

Last our method [12] of scheduling prioritizing is to assign tasks to each processor based on a higher number of their children. It means a task with more children would be scheduled earlier. In consideration of any task graph, the Number of Children (NC) for each task is calculated by Equation (3).

$$NC(t_i) = \begin{cases} 0 & \text{:if } t_i \text{ has no child} \\ \sum_{j=1}^{j=\text{number of outgoing edges from task } t_i} (1 + NC(t_j)) & \text{:as } t_j \text{ has the direct} \\ & \text{incoming edge from } t_i \end{cases} \tag{3}$$

Finally, all tasks are assigned to the processors (completeness) and each task is allocated only once (uniqueness). Our suggested prioritizing algorithm is illustrated with an example with attention to the task graph in Figure 1. The NC of each task is presented in Table 3 and the tasks are arranged in descending order based on their NC in Table 4. The EST for each task is shown in Table 5. By applying the above algorithm, a schedule would be produced.

Table 3: Relevant NC for each task of Figure 1

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
15	6	10	1	4	3	5	0	2	0	2	0	0	0	1	0

Table 4: Ordering tasks based on their NC

t_0	t_2	t_1	t_6	t_4	t_5	t_8	t_{10}	t_3	t_{14}	t_7	t_9	t_{11}	t_{12}	t_{13}	t_{15}
15	10	6	5	4	3	2	2	1	1	0	0	0	0	0	0

Table 5: EST for tasks

t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}
0	3	3	5	5	7	7	15	15	13	13	13	22	22	18	33

2.4 Prioritizing tasks based on the number of their children and height task in per group

New our method of scheduling prioritizing is to assign tasks to each processor based on a higher number of their children then tasks in per group whose equal number of their children prioritizing based on height task . It means a task with more children would be scheduled earlier and then tasks with number equal of children would be scheduled based on height them. In consideration of any task graph, the Number of Children (NC) for each task is calculated by Equation (3) and the height of a task would be calculated as Equation (2) [9]. A schedule producing algorithm based on the number of task children and height task is as follows:

1. Put tasks in a queue based on number of their children in descending order.
2. Separate tasks with the equal NC in a single group and perform steps 3 and 4 and 5 for all groups in order of higher NC until every group is empty.
3. Put tasks in select group in other queue in ascending order according to their height.
4. Produce a random number r between 1 and m ($m = \text{count processors}$).
5. Select the first task from the queue and allocate it to r^{th} processor and then delete it.

Finally, all tasks are assigned to the processors (completeness) and each task is allocated only once (uniqueness). Our suggested prioritizing algorithm is illustrated with an example with attention to the task graph in Figure 1. The NC of each task is presented in Table 3 and the tasks are arranged in descending order based on their NC in Table 4. The EST for each task is shown in Table 5. By applying the above algorithm, a schedule would be produced as shown in Figure 2.

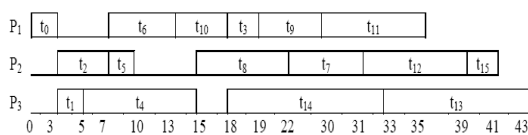


Figure 2: A schedule based on tasks NC and Hieght

3 The proposed algorithm

The genetic algorithm (GA) was developed by John Holland in 1975 [10] which is a search technique based on the principles of genetics and natural selection to find an optimal or sub-optimal solution. In GA, the term chromosome typically refers to a candidate

solution to a problem. GA allows a population composed of many chromosomes to evolve under specified selection rules to a state that maximizes the fitness (i.e., minimizes the cost function). GA is a method for moving from initial population of chromosomes to a new population by using a kind of genetic operators like crossover and mutation. Each chromosome consists of genes. The selection operator chooses those chromosomes in the population that will be allowed to reproduce new generation. Crossover exchanges subparts of two chromosomes and mutation randomly changes the values of some genes in the chromosome.

The new genetic algorithm introduced in this paper has following five phases:

3.1 The fitness value and initial population producing

The cost function of each schedule (i.e., the fitness of each chromosome) is selected as schedule length or TFT based on Equation (1). By repetition of the schedule producing algorithm based on the number of tasks children, the initial population will be produced.

3.2 Selection

The selection phase has two steps:

- 1) Applying a roulette wheel to select two chromosomes:

After ascending ordering of chromosomes based on their fitness, a roulette wheel series is constructed based on their fitness [1]. Hence, the chromosomes with lower TFT (best fitness), occupy more slots in the roulette wheel. In this way the possibility of selecting chromosomes with best fitness is higher. Then two chromosomes will be selected.

- 2) Applying a roulette wheel for selecting a task:

A roulette wheel is constructed for tasks based on their NC. A task with more children has more chance to be selected compared to a task with fewer ones.

The genetic operators like crossover, mutation and load balancing will be applied on the current generation to produce the next generation.

3.3 Crossover

A random number is produced between zero and one and if it is larger than the crossover rate or is equal to it, the crossover is done in the following way:

- 1) Two selected chromosomes in the selection phase are duplicated and the following operation is done on them to generate two new chromosomes.

- 2) All tasks would be chosen which have NC lower or equal to the NC of the selected task in the selection phase. For every processor of the first chromosome, the chosen tasks are exchanged with the other tasks in the peer processor in the second chromosome.

For example, the chromosomes C_1 and C_2 and the task t_{14} with NC value of 1 have been selected. During the crossover, the tasks which have NC lower or equal to 1 e.g. tasks $\{t_3, t_{14}, t_7, t_9, t_{11}, t_{12}, t_{13}, t_{15}\}$ are selected in both chromosomes and then are exchanged on their relevant peer processors as shown in Figure 3.

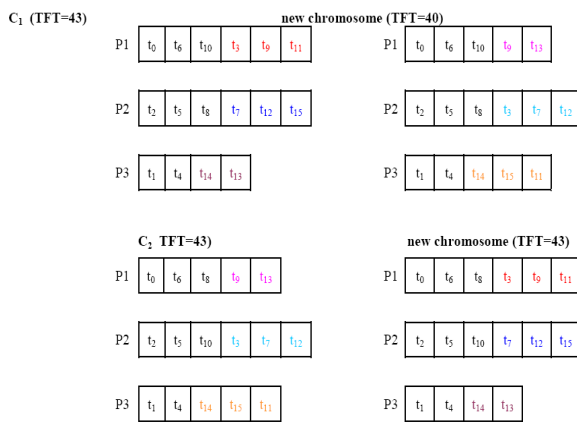


Figure 3: Applying crossover on C_1 and C_2 and producing two new chromosomes

3.4 Mutation

A random number between zero and one is produced and if it is larger than the mutation rate or is equal to it, the mutation operation is done in the following way:

new chromosome

3.5 Load balance

In this phase a new heuristic method called load balance is presented to reduce the TFT of chromosomes. The method involves following steps:

- 1) First, two selected chromosomes in selection phase are reduplicated and then the following operation is separately performed on two new chromosomes.
- 2) For one of the chromosomes from m processors, two processors which have the maximum and the minimum finishing time are selected (P_{max} and P_{min}). Then according to Equation (4), AVG is calculated as following:

$$AVG = (TFT(P_{max}) - TFT(P_{min})) / 2 \quad (4)$$

- 1) Two selected chromosomes in the selection phase are duplicated and then the following operation is done separately on them.
- 2) For the first chromosome, the selected task in the selected phase is exchanged with another task on different processor which has NC equal to it. The same operation is done on the second selected chromosome.

For example, the chromosome C_1 and the task t_{13} with NC value of zero are selected for the mutation. Another task from chromosome C_1 in different processor that has the NC equal to t_3 e.g. t_{15} is selected and two tasks t_{13} and t_{15} are exchanged as shown in Figure 4.

Lemma1. Since applying the mutation or crossover operators implies the uniqueness and completeness requirements have been met, after applying such operators, no task is missed and no task is added to the new chromosome. However, as all operators are based on tasks' NC, the tasks' execution precedence is met too.

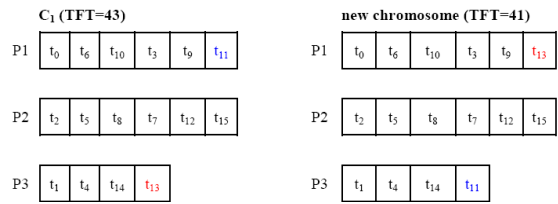


Figure 4: Applying mutation on C_1 and product a new chromosome

- 3) To balance the execution time of processors, task t_i is selected which is assigned to P_{max} and its execution time is equal or less than AVG. If such a task is not found, step 4 or load balance operation could not be performed.
- 4) Task t_i is deleted from P_{max} and then is added to P_{min} in a suitable place based on its NC in descending order, as all tasks' execution precedence is observed.

For example, the chromosome C and the task t_{15} are selected for the load balance operation. As shown in Figure 5, applying the load balance guarantees improvement of the fitness of chromosome C .

After load balance operation, the uniqueness and completeness requirements are met based on Lemma 1.

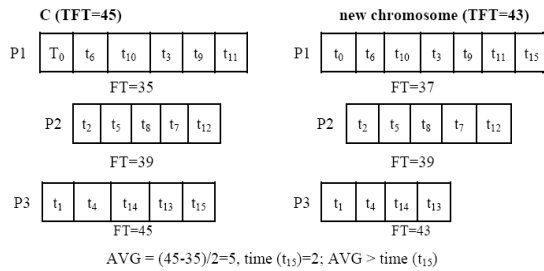


Figure 5: Applying load balance on C1 and producing a new chromosome

3.6 Reproduction

After applying all operators and producing new chromosomes, the former chromosomes along with new ones will be ordered based on their fitness and the next generation receives the most appropriate chromosomes (the chromosomes with lower TFT or best fitness) at the number of population size. Then the phases 3-2 to 3-6 will be repeated as the number of generations. Finally, the best suitable chromosome is the optimal or near-optimal schedule.

4 Simulations and Results

A range of simulations is done using the Visual Basic .Net version 2005 on a computer Pentium IV, having AMD processor 2.8 GHz, and 512 MB memory of RAM to evaluate our suggested algorithm.

Using our developed program - producing a random task graph automatically - 57 task graphs are created. Each graph could have 30, 70, or 90 tasks with task dependency percentage between 20 and 90 and the execution time for each task is random between 1 and 100 seconds. These graphs are scheduled on a multiprocessor system with the number of 3, 5, or 7 processors for five heuristics: min-min, max-min, duplex, MCT (Minimum Completion Time) and MET (Minimum Execution Time) [2] and for three genetic-based algorithms: Genetic Algorithm whose Priority is based on Task Height (GAPTH) [9] and last our proposed algorithm [12] and new our proposed algorithm. The results are averaged over multiple runs for each algorithm.

For three genetic-based algorithms, the crossover rate is set to 0.7 and the mutation rate is set to 0.05. Other parameters such as initial population size and the number of generations are selected similarly for genetic algorithms to perform the scheduling at the same conditions.

Table 6 The schedules for eight scheduling algorithms

Algorithms	Total Finish Time (TFT) (seconds)									TFT Mean (s)
	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	
min-min	992.3	1689	2303	902.3	1561	2231	908	1485	2285	1595
max-min	975.3	1669	2206	835.3	1466	2220	850.6	1436	2183	1579
Duplex	962.6	1669	2171	835.3	1466	2213	850.6	1436	2183	1572
MCT	988	1650	2148	836.6	1477	2271	858.6	1417	2177	1535
MET	966.6	1677	2164	835.3	1505	2268	890.3	1421	2178	1545
GAPTH	966.3	1634	2108	827.6	1447	2032	840.3	1410	2122	1487
Last our algorithm	881.6	1560	2052	727	1418	2141	800	1296	2049	1436
New our algorithm	873.3	1524	2011	720.9	1388	2107	777.5	1188	1991	1397

Table 6 shows the schedules and TFT mean for each eight 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 three above genetic algorithms is more than the other five heuristics obviously, and is quite similar, as only their initial population producing step is different, the step is calculated once. Table 7 illustrates the results of simulations with varying task dependency percentage. As shown here, if the number of tasks and processors, and the range of tasks' execution time are considered constant, then the higher percentage of the task dependency has better schedule for our developed new algorithm compared with the others genetic algorithm, named GAPTH and last our algorithm. The reason lies in the fact that the higher the task dependency is, the more number of children. Our scheduling new algorithm is more efficient than the other one as it acts based on NC.

Table 6 The schedules for eight scheduling algorithms

Algorithms	Total Finish Time (TFT) (seconds)									TFT Mean (s)
	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	
min-min	992.3	1689	2303	902.3	1561	2231	908	1485	2285	1595
max-min	975.3	1669	2206	835.3	1466	2220	850.6	1436	2183	1579
Duplex	962.6	1669	2171	835.3	1466	2213	850.6	1436	2183	1572
MCT	988	1650	2148	836.6	1477	2271	858.6	1417	2177	1535
MET	966.6	1677	2164	835.3	1505	2268	890.3	1421	2178	1545
GAPTH	966.3	1634	2108	827.6	1447	2032	840.3	1410	2122	1487
Last our algorithm	881.6	1560	2052	727	1418	2141	800	1296	2049	1436
New our algorithm	873.3	1524	2011	720.9	1388	2107	777.5	1188	1991	1397

5 Conclusions

The task scheduling problem in multiprocessor systems is an NP_Hard problem. Hence, using heuristic methods instead of classic ones, the optimal or near-optimal schedule would be achieved in an acceptable time. Due to the higher potential of genetic algorithms in solving the complex problems, they have been vastly acceptable in the heuristic methods. In this paper, a new genetic algorithm was presented for task scheduling in a multiprocessor system. In this algorithm, the priority of execution of tasks is based on the number of their children and for tasks with number of children equal in per group based on the height task , i.e., a task having more children will be scheduled earlier and then tasks with number of children equal in per group based on the height task. Our developed algorithm was compared to the genetic algorithm whose priority is based on number of children (last our algorithm)[12] , and to the five well-known heuristics and based on task height[9]. The results showed that our suggested new algorithm improves the achievement of the near-optimal schedule; however, the computation time of the three discussed genetic algorithms are quite the same.

References

1. Goldberg D. E.: Genetic Algorithms in Search, Optimization and Machine Learning, Reading, MA: Addison Wesley, (1989)
2. Braun T. D., Siegel H. J., Beck N. and et al.: A Comparison of Eleven Static Heuristic for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, vol. 61, pp. 810--837, (2001)
3. Rahmani A. M. and Resvani M.: A novel Static Task Scheduling in Distributed Systems by Genetic Algorithm using Simulated Annealing. 12th International CSI Conference, Iran, p. 83, (2007)
4. Bouffard V., Ferland J. A.: Improving simulated annealing with variable neighborhood search to solve the resource-constrained scheduling problem. *Journal of Scheduling*, Vol. 10(4), pp. 375--386, (2007)
5. Silva M. L. and Porto S. C. S.: An Object-Oriented Approach to a Parallel Tabu Search Algorithm for the Task Scheduling Problem. *Proceedings of the 19th International Conference of the Chilean Computer Science Society*, p. 105, (1999)
6. Shenassa M. H. and Mahmoodi M.: a novel intelligent method for task scheduling in multiprocessor systems using genetic algorithm. *journal of Franklin institute*, Elsevier, (2006)
7. Yoo M. and Gen M.: Scheduling algorithm for real-time tasks using multiobjective hybrid genetic algorithm in heterogeneous multiprocessors system. *Computers and Operations Research*, Vol. 34(10), P. 3084--3098, (2007)
8. Zheng S., Shu W. and Dai S.: Task Scheduling Model Design Using Hybrid Genetic Algorithm. in *Proceedings of the First International Conference on Innovative Computing, Information and Control*, Vol. 3, pp. 316--319, (2006)
9. Hou E. S. H., Ansari N. and Ren H.: A Genetic Algorithm for Multiprocessor Scheduling. *IEEE trans. on parallel and distributed systems*. vol. 5, no. 2, pp. 113--120, Feb. (1994)
10. Holland J. H.: *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, (1975)
11. Zafarani Moattar E., Rahmani A.M., Feizi Derakhshi M.R., "Job Scheduling in Multi Processor Architecture Using Genetic Algorithm", 4th IEEE International conference on Innovations in Information Technology, dubai, pp. 248-251, (2007)
12. Abdeyazdan M., Rahmani A.M., " 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.

A parallel algorithm based on simulated annealing for land use zoning plans

M. Suárez¹, I. Santé¹, F. F. Rivera², R. Crecente¹, M. Boullón¹, J. Porta³, J. Parapar³, and R. Doallo³

¹Land Laboratory, University of Santiago de Compostela, Lugo, Spain

²Dept. of Electronics and Computing, University of Santiago de Compostela, Santiago de Compostela, Spain

³Computer Architecture Group, University of A Coruña, A Coruña, Spain

Abstract—*There is an increasing demand for tools which support the land use planning process and one of the most complex tasks of this process is the design of a land use zoning map. With this aim an algorithm based on simulated annealing has been designed to optimize the delimitation of land use categories according to suitability and compactness criteria. The high number of plots involved in a land use plan leads to high computational costs. Two parallel versions were implemented. The first one improve the final solution using different parameters in parallel. The second one gets advantage of the spatial parallelism. Results on a real case of study show that the solutions provided by our algorithms are similar to the solution provided by experts, but much faster and with less effort. The parallel versions of the code present good results in terms of the quality of the solution and speed-up.*

Keywords: Land use planning, parallel simulated annealing, GIS

1. Introduction

The development of a land use plan is long and laborious, requiring a great effort on the part of public administrations and technical teams to achieve a good solution. As a result, there is an increasing demand for tools which support the planning process and one of the most complex tasks of this process is the design of a land use zoning map. The design of a land use zoning map can be formulated in terms of an optimization problem in which each plot is allocated to the best category according to certain criteria and constraints. These criteria always include the land suitability for the land uses of a land category (e.g., [1], [2]) and some authors also consider spatial criteria, especially the compactness of the regions allocated to one single category (e.g., [3], [4], [5]). Due to the fact that the number of plots involved in a municipal land use plan is usually large, the search of the optimal solution using algorithms such as integer programming is unfeasible. It is, therefore, necessary to turn to heuristic algorithms capable of achieving near-best solutions in a reasonable time [6] [7] [8]. In particular, good results have been obtained using the simulated annealing technique (e.g., [9], [10], [11], [4], [12], [13]). Most of these algorithms operate on a regular raster grid. Land use zoning based on a regular grid is found to be unrealistic as it may

lead to a single-land use plot allocated to several categories or to a group of very different plots allocated to a single category. In addition to this, the planning laws in the study area require land use zoning based on cadastral plots.

The large number of plots involved in a municipal land use plan leads to high computational costs in order to run a number of iterations enough for exploring the complete search space. The use of different parallelization strategies has been considered in order to reduce the execution time and to improve the results of the algorithm. Many proposals for parallelization can be found in the literature [14], [15].

This study proposes a parallel algorithm based on simulated annealing for land use zoning that uses an irregular spatial structure based on a cadastral parcel map. This algorithm was applied to land use zoning in the municipality of Guitiriz, located in Galicia (N.W. of Spain) as a case of study. The paper is structured as follows: Section 2 defines the characteristics of the optimization problem. Section 3 describes the pre-processing stage. Section 4 describes the design of the simulated annealing algorithm. Section 5 is devoted to introduce the implementation of the parallel versions of the algorithm. In section 6 experimental results are discussed. And finally, some conclusions and ideas for future work are given in Section 7.

2. Problem statement

Land use planning laws define a set of land use categories and the restrictions enforced to each category. For some categories, their spatial allocation is completely and uniquely determined by legal restrictions. We will refer to this group of categories as fixed categories. In the case of Galicia, the fixed categories include the water, coast, infrastructure and heritage protection land. The non-fixed categories correspond to the agricultural, forestry, natural space and urban land.

Consequently, two stages can be distinguished in the design of a land use zoning map: the application of law restrictions for the delimitation of fixed categories and the decision making by planners for the allocation of non-fixed categories. For the first stage a pre-processing module has been developed in which the fixed categories are allocated applying the planning laws by means of geometric operations (buffers, intersections, differences...).

In the second stage planners must delimitate the non-fixed categories using their expert knowledge. An heuristic algorithm based on simulated annealing has been designed to facilitate this task. At this point it is important to note that laws and experts advise that the process of spatial allocation should take into account the current boundaries of the existing plots in the municipality, i. e., a plot should not be divided in several parts with different categories. Therefore the problem is to distribute N plots among C different non-fixed categories addressing two objectives, based on experts' criteria: maximization of the overall suitability of the plots to the categories allocated to them and maximization of the compactness (and hence minimization of the fragmentation) of the resultant land use patches. Land use patches are defined as the polygons resulting from the union of plots assigned to the same category. This optimization is subject to the constraints that the total area allocated to each non-fixed category cannot exceed certain minimum and maximum values set by the planner.

The relative importance of both suitability and compactness criteria varies depending on the target land category. For example, the compactness is basic for the forestry land category, whereas in the natural space land the importance of the compactness is low. For this reason the planner must be able to assign different weights to each criterion in each category.

3. Pre-processing stage

The problem requires three types of data which are read in the pre-processing stage: characteristics of each plot, parameters for the allocation of each category and geometric elements to define fixed categories. The characteristics of each plot include its geometry, initial category and a suitability score for each non-fixed category. In addition, the parameters for the allocation of each category include the maximum and minimum area, and the weights for the suitability and compactness criteria for each category.

The elements that define the fixed categories correspond to layers of geometric elements like rivers, roads, archaeological sites. They can delimitate the fixed categories in different ways. The first one is to allocate directly these elements to a specific fixed category. For example, the archaeological sites are included directly in the heritage protection land. This procedure also allows to allocate a category to areas that, because of their singularity, must be delimited by and expert or a specific algorithm. The other issue is the delimitation of a buffer over the geometric elements at a certain distance established by the law. An example is the protection area for roads. The result of these procedures is a map of plots in which the fixed categories are delimited, so the plots allocated to these categories are not considered in the simulated annealing algorithm. The pre-processing stage allows that these calculations, most of them involving computationally expensive geometric operations (e.g. intersections), are run

just once. These operations have been implemented using the JTS Topology Suite [16] library for spatial analysis operations and the SEXTANTE framework [17].

Besides, in the pre-processing stage the conditioning of the algorithm input data is carried out. The calculation of the compactness based on land use patches requires the knowledge of the adjacent plots to each one of them, called neighbours and the length of the border that they share. With the aim of speeding up this calculation, in the pre-processing stage a spatial indexing of the plot map is used, so the index query provides the list of plots candidates to be neighbours, thereby reducing the number of processed plots from several tens of thousands to a few tens. Choosing the right data structure to store the neighbours and the length of its borderline is an important issue since this information is accessed often by the algorithm, so it is important to minimize the time to access it. As the number of neighbours of each plot can be different, two unidimensional arrays are used to store the list of neighbours: an array of neighbours and an index array. The i -th entry of the index array stores the position in which the first neighbour of the i -th plot is stored in the array of neighbours, where $i = 1 \dots N$. The neighbours of each plot are stored consecutively in the array of neighbours. Figure 1 shows an example of these two arrays. In this example neighbours of plot P2 are P1, P3, P6 and P7, and they are stored from position 4 of the array of neighbours. Note that 4 is the value of the second entry in the index array. This structure presents low latency in its access.

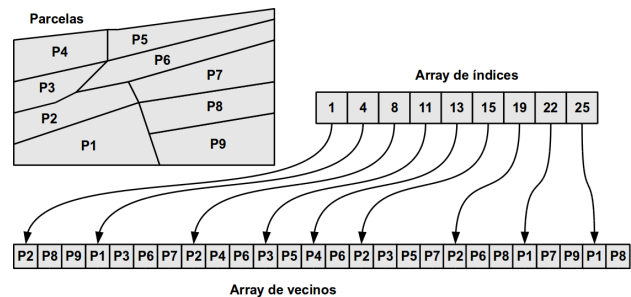


Fig. 1: Arrays used to store the information about the neighbourhood.

4. Simulated annealing algorithm

The simulated annealing algorithm [18] is a heuristic to intensively optimize an objective function ruled by a parameter called temperature T that is used to control the thoroughness of the search for the optimum. The basic procedure is as follows: (1) given the current configuration of the system, a trial configuration is generated by a method that includes some element of chance. (2) The value of the objective function for the trial configuration, E_t , is compared with the value of the objective function for the current configuration,

E_c . If E_t is better than E_c , the trial configuration becomes the current configuration, otherwise the trial configuration is adopted as the next current configuration according to the Boltzmann probability distribution: $e^{(E_c - E_t)/T}$. (3) For each value of temperature, the system is allowed to explore the configuration space for a number of iterations. The value of T is then reduced, so that better E values are favoured and the loop starts from step 1. (4) The algorithm terminates upon satisfaction of some appropriate stop condition.

In our case, at the beginning of the process an initial random solution is generated that satisfies the constraints of maximum and minimum area for each category.

4.1 The objective function

The objective function E combines two subobjectives: maximization of land suitability and maximization of compactness. These subobjectives are combined linearly:

$$E = W_c \times compactness + W_s \times suitability \quad (1)$$

where W_s and W_c are defined by the planner and are normalized so that the summation of both weights must be 1. The subobjective functions are normalized to the range [0, 1].

Suitability is calculated as the weighted average of the suitability for each category. Suitability for a category is obtained from the average of the suitability of the plots allocated to that category, weighted by the area of each plot and normalized by the total area assigned to the category:

$$Suitability = \sum_{i=1}^C w_i \left(\frac{\sum_{j=1}^{N_i} S_{ij} \times a_{ij}}{\sum_{j=1}^{N_i} a_{ij}} \right) \quad (2)$$

where w_i is the weight of the i -th category, N_i is the number of plots allocated to the i -th category, s_{ij} is the suitability of the j -th plot allocated to the i -th category, and a_{ij} is the area of the j -th plot allocated to the i -th category.

Compactness can be defined in different ways. In our proposal two different functions are considered: one based on patches, which are groups of adjacent plots with the same category, and the other one based on categories, where the plots are grouped into categories. For the compactness based on patches the function is defined as:

$$Compactness = 4\pi \sum_{i=1}^C w_i \left(\frac{\sum_{j=1}^{NP_i} \frac{A_{ij}}{P_{ij}^2}}{NP_i} \right) \quad (3)$$

where NP_i is the number of patches of the i -th category, A_{ij} and P_{ij} are the area and perimeter of the j -th patch of the i -th category, respectively. This formula is based on the fact that, for a given area value, the so called circularity is maximized by a circle (and the maximum is 1) [19]. The compactness function based on categories:

$$Compactness = 4\pi \sum_{i=1}^C w_i \left(\frac{\sum_{j=1}^N a_{ij}}{\sum_{j=1}^N p_{ij}^2} \right) \quad (4)$$

where p_{ij} is the perimeter of the j -th plot allocated to the i -th category. Note that this function has clearly a lower computational cost than (3) because it avoids the computation of patches.

4.2 Computational issues

Computing the objective function for a trial solution E_t is done by calculating the variation of E due to the change of the category of the involved plot instead of calculating the overall suitability and compactness for the whole plot map. The new value of the suitability subobjective is calculated by subtracting the area-weighted suitability of the changed plot for the old category and by adding the area-weighted suitability of the changed plot for the new category.

In the case of the compactness function based on patches, the compactness score is computed from the area and perimeter of each patch. In the calculation of the compactness of the new category three situations can be distinguished; i) a new patch is generated, ii) the area of an existing patch increases, or iii) several patches are merged. In the calculation of the compactness of the old category also other three situations can happen; i) a patch disappears, ii) the area of an existing patch decreases, or iii) a patch is divided into several patches. The identification and management of these situations is performed as follows. In the case of the new category, if no neighbour plot has the new category, a new patch is generated that has the area and perimeter of the changed plot. If a neighbour plot whose category is the same as the new one is found, the patch is reconstructed from the changed plot using the neighbour patch ID. This case can correspond to any of the following two situations: the area of an existing patch simply has increased or several patches have been merged. In the latter case other neighbour plots with the new category and different patch ID will be found, so this patch ID is removed since that patch has been merged with the previous reconstructed patch. In the case of the old category, it is considered that the patch has disappeared unless a neighbour plot whose category coincides with the old category is found. From this neighbour plot, a patch is reconstructed by a recursive flooding algorithm and the plots that constitute the reconstructed patch are identified. If other neighbour plot that has the old category, and not included in the reconstructed patch is found, a new patch is generated from this neighbour plot because the old patch has been fragmented.

The compactness function based on patches presents an interesting issue when several patches are merged. Since the overall compactness is the average of the compactness of each patch, when two patches with relatively high compactness are merged, the resulting patch often has lower compactness and consequently generates lower overall compactness. It is important to deal with these situations because merging patches produces benefits at long term. Therefore a mechanism to promote the conservation of changes that

merge patches has been introduced. This mechanism consists on increasing the temperature of the Boltzmann test by a certain factor when the number of patches of the new solution is lower than the number of patches of the old solution in order to increase the probability of acceptance of the new solution. A multiplying factor is introduced and tuned by the planner to control this mechanism.

In the case of the compactness function based on categories, the new value of this subobjective is calculated by modifying the area and the perimeter of the old and new categories. The area is modified by subtracting the area of the changed plot from the compactness score of the old category and by adding it to the compactness score of the new category. The new values of perimeter for each category are obtained by comparing the old and new categories of the changed plot with the category of their neighbour plots.

4.3 The annealing schedule

The parameters of the annealing schedule must be defined by the planner. In general, it is recommended that the initial value of T ensure that about 80% of trials are successful at this stage; this value will depend on both the way in which the objective function varies with configuration, and the configuration generating scheme, and must be identified by trial and error for each problem. The heat balance condition, that is, the number of iterations executed at each temperature, was approximately twice the number of plots and each reduction of T was affected by multiplying it by a constant factor, which was 0.95 by default. The stop condition of the algorithm is the number of temperatures established by the planner, which was set to 200 by default.

Figure 2 shows the evolution of compactness and suitability with the temperature using the compactness function based on categories. The compactness function based on patches has similar behaviour. In shown case the compactness values and the variations of these values are very low, so a higher weight must be assigned to the compactness subobjective.

In Figure 3 note that the compactness function based on patches tends to generate a greater number of patches. In this figure each category is identified with a different grey level.

5. Parallel simulated annealing

The computational cost of the algorithm is high due to the large number of plots and the implicit nature of the problem. To get a more practical algorithm, the execution time has to be reduced, and the solution lies in its parallelization. Two strategies to parallelize the simulated annealing are proposed: parameter parallelization and spatial parallelization. A third possible strategy was also considered based on the parallelization of the computation of the objective function. However we found out that it is not efficient, mainly because the low computational cost of the objective

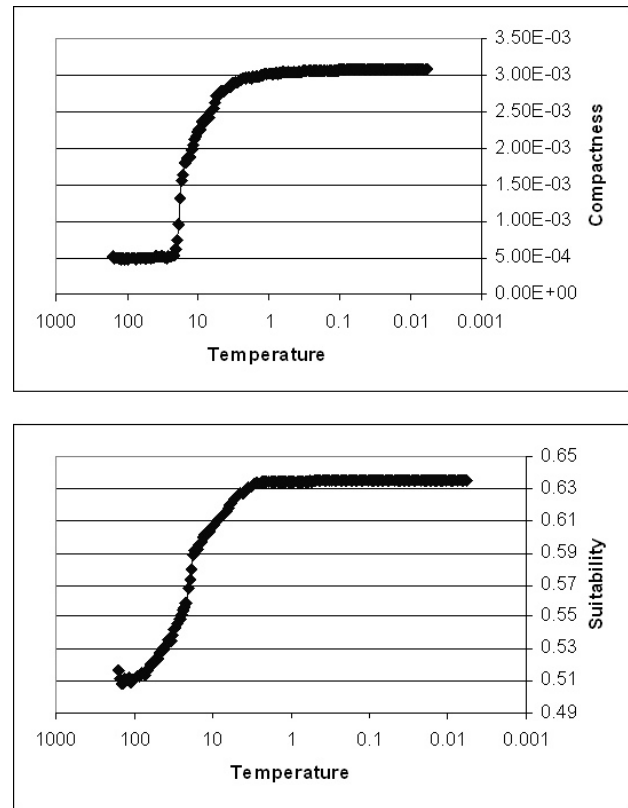


Fig. 2: Compactness and suitability evolution with the temperature for compactness function based on categories or patches.

function as, according to our proposal, only changes caused by the changes in the category of a single plot are taken into account.

5.1 Parameter parallelization

The identification of the optimal values of the parameters that guide the annealing is key issue to find the best solution. Running the algorithm in parallel with different parameter values helps in the search of the optimal values. These parameters are: the initial temperature, the number of iterations for each temperature, the cooling coefficient, the initial solution and the weights of both subobjectives. The parallelization uses as many processes as the number of different initial temperatures in this case of study.

5.2 Spatial parallelization

The spatial parallelization implies that each process run the algorithm in a particular geographic zone of the study area. The plot map is partitioned into groups of plots that are completely surrounded by plots allocated to the fixed categories, that is, by plots excluded from the simulation. In this way, there are no borderline interactions among zones. Each of these isolated groups of plots is called a cluster. The algorithm identifies the clusters, from the plot map by a

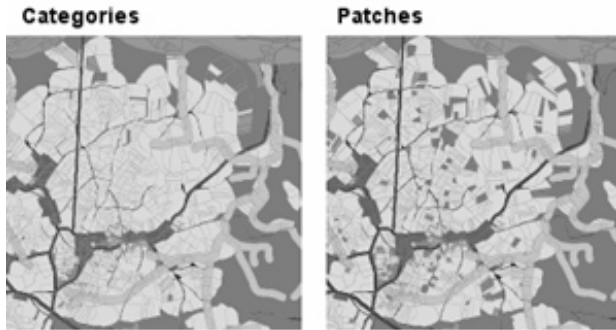


Fig. 3: Land zoning maps obtained with both objective functions; the first one uses the compactness function based on categories and the second one the compactness function based on patches.

pre-processing stage, using a flooding algorithm. In order to balance the computational load, the clusters are distributed among the processes so that the number of plots in each process is as similar as possible to the others.

The execution of each process is practically independent from the rest of them. The only common data accessed by all the processes is the value of the total area allocated to each category. Note that this area is constrained between a certain minimum and a maximum. Therefore changes of the category of a plot that results in a total area for a category exceeding the minimum and maximum values cannot be done. Therefore this constraint must be checked continuously using mutual exclusion operations.

6. Case of study

As a case study, a part of a Galician municipality called Guitiriz, was considered, it consists of 36,803 plots. After the pre-processing stage 34,000 polygons do not have a fixed category, so it must be taken into account in the simulated annealing stage. The plot suitability for each category and the total area to be allocated to each category were obtained from previous studies [20]. All performance tests were executed in a system with 2 processors Intel Xeon E5440 2.83GHz, 4 cores each and 16GBs of shared memory.

6.1 Parameter parallelization

Figure 4 shows compactness and suitability dependence with the initial temperature and with different weights for both subobjectives labelled by the values of w_c and w_s respectively.

There is not a clear trend in the influence of initial temperature on objective function values, so the evaluation of a wide range of temperature values is important to find the best value of this parameter. Figure 5 shows the results obtained by running the algorithm three times with the same parameters but with different initial solutions. Differences between the solutions obtained with different initializations

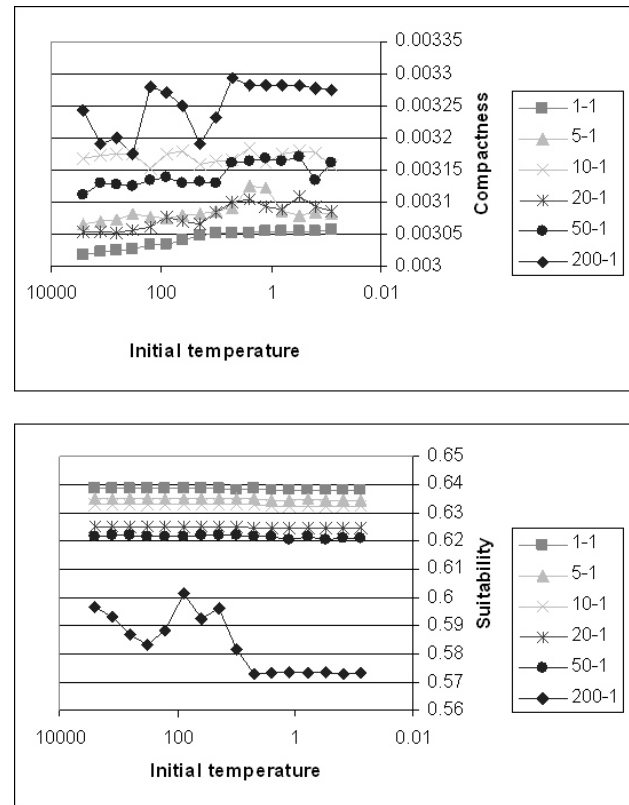


Fig. 4: Influence of the initial temperature on compactness and suitability by using the compactness function based on categories and different weights for the subobjectives.

but the same parameters are similar to the differences obtained by varying the weighting of the subobjectives.

6.2 Spatial parallelization

In figure 6 the influence of the initial temperature and the number of processes in the algorithm are shown. The increase in the number of processes decreases the suitability value but does not influence the compactness value. Note that influence on the initial temperature is very low.

Figure 7 shows the speedup for 7 different situations that are labelled as: the kind of algorithm (by patches or by categories), the values of w_c and w_s , and the initial temperature respectively.

The highest speedups are obtained with the compactness function based on patches and with the compactness function based on categories when only the compactness subobjective is optimized, because in these cases each process manages only local data.

6.3 The influence of the compactness

The compactness based on categories provides solutions with a number of patches quite lower than the compactness metric based on patches. Table 1 shows the number of patches in the final solution by using different compactness

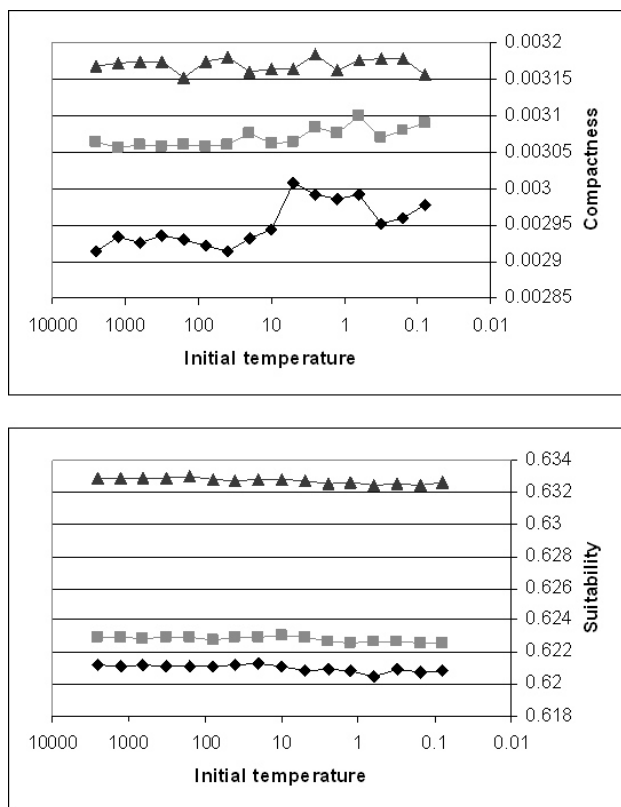


Fig. 5: Influence of the initial solution.

functions, different weights for subobjectives and different values for the temperature multiplier. Note that an increase on the value of the temperature multiplier factor reduces the number of patches. Anyway this number is higher than the number of patches generated by the compactness based on categories. The increase of the suitability subobjective weighting when using compactness function based on patches reduces the number of patches, because the spatial distribution of suitability presents a certain compactness by itself.

Table 1: Number of patches

Compactness	Wc	Ws	T multiplier	Patches number
Categories	0	1		7786
Categories	1	1		6361
Categories	5	1		5677
Categories	10	1		5496
Categories	20	1		5369
Categories	50	1		5235
Categories	1	0		4786
Patches	1	0	1	8842
Patches	1	0	100	7698
Patches	1	1	100	7193

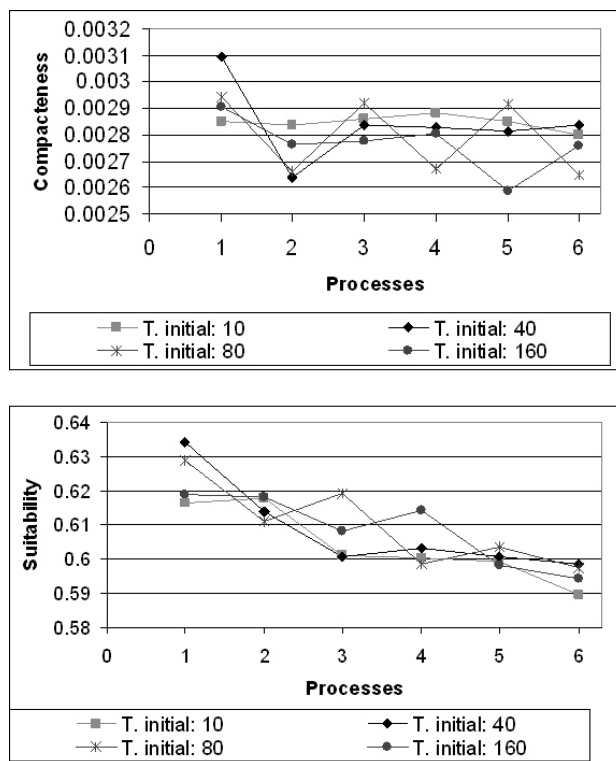


Fig. 6: Compactness and suitability values for each number of processes (the objective function uses the compactness function based on categories).

6.4 Comparison with handmade planning

In order to evaluate the solutions provided by the algorithm, these solutions have been compared to the land use zoning map designed by technicians for the municipal land use plan of Guitiriz. The overlap area was measured as percentage of the total area of the technical solution. For compactness based on categories the coincidence was: agricultural 78%, forestry 68%, natural space 91% and urban 49%. For compactness based on patches the coincidence was: agricultural 78%, forestry 67%, natural space 87% and urban 49%. This results show a good matching for agricultural, forestry and natural space categories. The causes of the worst matching of urban category are the aesthetic and architectural criteria used by technicians in urban planning, which are not considered in the algorithm. However, the global suitability for the land zoning map designed by technicians is 0.52 and for the land zoning maps provided by the algorithm is around 0.62, so a significant improvement is achieved.

7. Conclusions

In this paper we deal with the problem of land use planning. Our proposal is to solve the delimitation of land use categories issue, that is frequently the stage of the

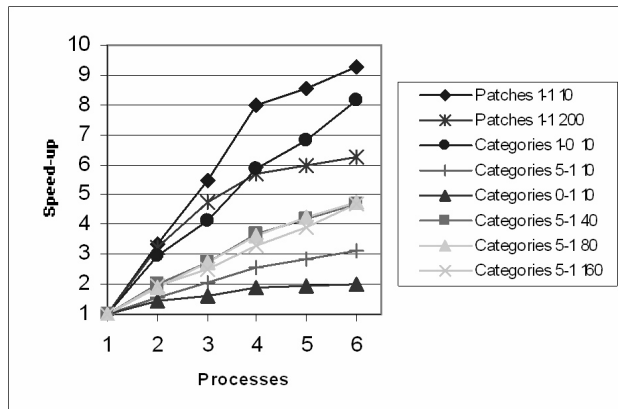


Fig. 7: Performance of the spatial Geographical parallelization.

whole process that is the bottleneck in practice. After a preprocessing stage, a simulated annealing based heuristic is used to efficiently solve the problem. An objective function that is a linear combination of two factors: the suitability and the compactness is introduced. The quality of the results on real situations are comparable to those obtained by experts. However the best parameters that rule the annealing can not be easily established prior to the execution. We used this feature to parallelize the algorithm by running the sequential code in several processes using different parameters. The benefits of this approach are mainly in the improvement on the quality of the final result. In addition, a spatial parallel implementation is proposed in which the geographical zone of study is partitioned into a number of so called clusters that can be processed in parallel. Appropriate mechanisms to share the information among the processes have been implemented. The efficiency of this second parallel implementation was validated in a real case of study. Both parallel proposals are orthogonal, so they can be applied simultaneously.

As a future work, one of the most immediate improvements is to study other kind of functions for the evaluation of the compactness criteria. The optimization of other spatial metrics such as connectivity is also interesting, especially for the case of the natural space category in order to design ecological networks.

8. Acknowledgment

This work is included in the project named "Geographical Information Systems for Urban Planning and Land Management using Optimization Techniques on Multicore Processors" with code 08SIN011291PR, and Consolidation of Competitive Research Groups (ref. 2010/06 and 2010/28), funded by the Galician Regional Government, Spain. And by Ministry of Education and Science of Spain under contract TIN 2007-67537-C03.

References

- [1] J. R. Eastman, W. Jin, P. A. K. Kyem, and J. Toledano, "Raster procedures for multi-criteria/multi-objective decisions," *Photogrammetric Engineering and Remote Sensing (PE&RS)*, vol. 61, no. 5, pp. 539–547, May 1995.
- [2] R. G. Cromley and D. M. Hanink, "Scale-independent land-use allocation modeling in raster gis," *Cartography and Geographic Information Science*, vol. 30, pp. 343–350(8), 1 October 2003.
- [3] J. C. J. H. Aerts, E. Eisinger, G. B. M. Heuvelink, and T. J. Stewart, "Using linear integer programming for multi-site land-use allocation," *Geographical Analysis*, vol. 35, no. 2, pp. 148–169, 2003.
- [4] D. Nalle, J. Arthur, and J. Sessions, "Designing compact and contiguous reserve networks with a hybrid heuristic algorithm," *Forest Science*, vol. 48, no. 1, pp. 59–68(10), February 2002.
- [5] T. J. Stewart, R. Janssen, and M. van Herwijnen, "A genetic algorithm approach to multiobjective land use planning," *Comput. Oper. Res.*, vol. 31, pp. 2293–2313, December 2004.
- [6] K. B. Matthews, S. Craw, and A. R. Sibbald, "Implementation of a spatial decision support system for rural land use planning: integrating gis and environmental models with search and optimisation algorithms," *Computers and Electronics in Agriculture*, vol. 23, pp. 9–26, 1999.
- [7] T. Cay and F. Iscan, "Fuzzy expert system for land reallocation in land consolidation," *Expert Systems with Applications*, vol. 38, no. 9, pp. 11 055 – 11 071, 2011.
- [8] J. Porta, J. Parapar, G. L. Taboada, R. Doallo, F. F. Rivera, I. Santé, M. Suárez, M. Boullón, and R. Crecente, "A java-based parallel genetic algorithm for the land use planning problem," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2011*, Dublin, Ireland, July 12-16, July 12-16 2011, accepted.
- [9] J. C. J. H. Aerts and G. B. M. Heuvelink, "Using simulated annealing for resource allocation," *International Journal of Geographical Information Science*, vol. 16, no. 6, pp. 571 – 587, 2002.
- [10] M. Boyland, J. Nelson, and F. L. Bunnell, "Creating land allocation zones for forest management: a simulated annealing approach," *Canadian Journal of Forest Research*, vol. 34, no. 8, pp. 1669–1682, 2004.
- [11] E. Martínez-Falero, I. Trueba, A. Cazorla, and J. L. Alier, "Optimization of spatial allocation of agricultural activities," *Journal of Agricultural Engineering Research*, vol. 69, no. 1, pp. 1 – 13, 1998.
- [12] I. Santé-Riveira, R. Crecente-Maseda, and D. Miranda-Barros, "Gis-based planning support system for rural land-use allocation," *Computers and Electronics in Agriculture*, vol. 63, no. 2, pp. 257 – 273, 2008.
- [13] S. K. Sharma and B. G. Lees, "A comparison of simulated annealing and gis based mola for solving the problem of multi-objective land use assessment and allocation," in *17th International Conference on Multiple Criteria Decision Analysis*, Whistler, Canada, August 2004, pp. 6 –11.
- [14] E. Onbasoglu and L. Ozdamar, "Parallel simulated annealing algorithms in global optimization," *Journal of Global Optimization*, vol. 19, pp. 27–50, 2001, 10.1023/A:1008350810199.
- [15] Z. J. Czech, *Parallel and Distributed Computing*. InTech, January 2010, ch. A Parallel Simulated Annealing Algorithm as a Tool for Fitness Landscapes Exploration, pp. 247–272.
- [16] M. Davis and J. Aquino, "Jts topology suite technical specifications," Vivid Solutions, Tech. Rep., 2003. [Online]. Available: [http://www.vividsolutions.com/JTS/bin/JTS Technical Specs.pdf](http://www.vividsolutions.com/JTS/bin/JTS%20Technical%20Specs.pdf)
- [17] V. Olaya, "Sextante programming guide," Sextante, Tech. Rep., 2011. [Online]. Available: <http://www.sextantgis.com>
- [18] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, 1953.
- [19] R. S. Montero and E. Bribeasa, "State of the art of compactness and circularity measures," *International Mathematical Forum*, vol. 4, no. 27, pp. 1305–1335, 2009.
- [20] I. Santé, R. Crecente, M. Boullón, and D. Miranda, *Spatial Decision Support for Urban and Environmental Planning. A Collection of Case Studies*. Malaysia: Arah Publications, 2009, ch. Optimising land use allocation at municipal level by combining multicriteria evaluation and linear programming, pp. 33–60.

Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs

Atanas Radenski

School of Computational Sciences, Chapman University, Orange, California, USA

radenski@chapman.edu <http://www.chapman.edu/~radenski>

Abstract – While merge sort is well-understood in parallel algorithms theory, relatively little is known of how to implement parallel merge sort with mainstream parallel programming platforms, such as OpenMP and MPI, and run it on mainstream SMP-based systems, such as multi-core computers and multi-core clusters. This is unfortunate because merge sort is not only a fast and stable sort algorithm, but it is also an easy to understand and popular representative of the rich class of divide-and-conquer methods; hence better understanding of merge sort parallelization can contribute to better understanding of divide-and-conquer parallelization in general. In this paper, we investigate three parallel merge-sorts: shared memory merge sort that runs on SMP systems with OpenMP; message-passing merge sort that runs on computer clusters with MPI; and combined hybrid merge sort, with both OpenMP and MPI, that runs on clustered SMPs. We have experimented with our parallel merge sorts on a dedicated Rocks SMP cluster and on a virtual SMP cluster in the Amazon Elastic Compute Cloud. In our experiments, shared memory merge sort with OpenMP has achieved best speedup. We believe that we are the first ones to concurrently experiment with - and compare - shared memory, message passing, and hybrid merge sort. Our results can help in the parallelization of specific practical merge sort routines and, even more important, in the practical parallelization of other divide-and-conquer algorithms for mainstream SMP-based systems.

Keywords: Parallel merge sort, OpenMP, MPI, SMP, Cluster computing, cloud computing

1 Introduction

Merge sort is an efficient divide-and-conquer sorting algorithm. Because merge-sort is easier to understand than other useful divide-and-conquer methods, it is often considered to be a typical representative of such methods, and frequently used to introduce the divide-and-conquer approach itself [3, Ch 2].

Intuitively, merge sort operates on an array of n objects as follows: (1) if $n > 1$, divide the array into two sub-arrays of about half the size each; (2) apply merge sort on each sub-

array; (3) merge the two sorted sub-arrays from step 2 into one sorted array. For small arrays, some implementations switch from recursive merge sort to non-recursive methods, such as insertion sort – an approach that is known to improve execution time. (Fig. 1 in Section 2.1 outlines a serial merge sort implementation in C.)

The average complexity of merge sort is $O(n \log n)$ [7], the same as quick sort and heap sort. In addition, best-case complexity of merge sort is only $O(n)$, because if the array is already sorted, the merge operation perform only $O(n)$ comparisons; this is better than best case complexity of both quick sort and heap sort. The worst case complexity of merge sort is $O(n \log n)$ [7], which is the same as heap sort and better than quick sort. However, classical merge sort uses an additional memory of n elements for its merge operation (the same as quick sort), while heap sort is an in-place method with no additional memory requirements.

The average/best/worst asymptotic complexity of merge sort is at least as good as the corresponding average/best/worst asymptotic complexity of heap sort and quick sort; despite of this, merge sort is often considered to be slower than the other two in practical implementations. On the positive side, merge sort is a stable sort method, in contrast to quick sort and heap sort, which fail to maintain the relative order of equal objects. The practical performance of merge sort is known to improve with recursion removal and cache memory utilization [8].

The focus of this paper is not on efficiency improvements that are specific to merge sort. Instead, we regard recursive merge sort as a typical and well-understood representative of the divide-and-conquer approach. We use merge sort as a test bed to explore parallelization schemes that may possibly apply without significant changes to other divide-and conquer methods.

Merge sort parallelization is well-studied in theory. For example, Cole [2] describes a $O(\log n)$ parallel merge sort algorithm for a CRW PRAM (an abstract machine which neglects synchronization and communication), while Cormen et al outline another $O(\log n)$ parallel merge sort for abstract comparison networks [3, Ch. 27].

In contrast to theory, little is known of how to implement parallel merge sort on mainstream architectures (such as standalone and clustered Symmetric Multiprocessing Systems, SMPs), by means of mainstream shared memory and

message passing platforms (such as OpenMP [13] and MPI [12]). Our goal in this paper is to provide a better understanding in this direction.

We choose OpenMP to parallelize merge sort on SMPs and MPI to parallelize it on clustered systems. We choose OpenMP to implement shared-memory merge sort on SMPs because (i) OpenMP is standardized and comes ready-to-use with contemporary C/C++ compilers, including compilers that are freely available; (ii) OpenMP is easier to use than various thread libraries because it supports a higher level parallel programming model; (iii) OpenMP is can work on a wider number of shared-memory computers as opposed to other interesting yet less available higher-level frameworks, such as UPC [4] and Orio [9]. We choose MPI to implement message-passing merge sort on computer clusters because (i) MPI is implemented for a broad variety of architectures, including implementations that are freely available; (ii) MPI is well documented; (iii) MPI has grown much more popular than alternative platforms, such as PVM [5]. Finally, our preference for an implementation language is ANSI C because (i) C is fast and available on virtually any platform; (ii) C can be used to implement merge sort versions with both OpenMP and MPI, including a hybrid implementation of parallel merge sort, based on both OpenMP and MPI (see Section 2.3).

In the rest of this paper, we describe parallel merge-sort algorithms with OpenMP and MPI, and evaluate their performance (Section 2); then we offer conclusions (Section 3). Section 2.1 is devoted to a shared memory OpenMP implementation of merge sort, while Section 2.2 delivers a message-passing merge sort with MPI. Section 2.3 is focused on a hybrid parallel sort that combines both OpenMP and MPI. Section 2.4 evaluates and compares the performance of the three parallel merge sorts as measured on a dedicated SMP cluster. In addition, Section 2.5 describes experience with the same parallel merge sorts on AWS, the Amazon cloud computing platform [1] and provides performance evaluation accordingly.

2 Recursive Merge Sort Parallelization and Evaluation

Recursive merge sort is a typical and well-understood divide-and-conquer algorithm (Fig. 1).

```
void mergesort_serial(int a[], int size, int temp[]) {
    if (size < SMALL) { insertion_sort(a, size); return; }
    mergesort_serial(a, size/2, temp);
    mergesort_serial(a + size/2, size - size/2, temp);
    merge(a, size, temp);
}
```

Fig. 1. Serial recursive merge sort in C. It sorts an array a using additional array $temp$ of the same size as a

We design parallel versions of this algorithm not as much for the sake of merge sort parallelization alone, but to also hopefully provide insights into parallelization of divide-and-conquer algorithms in general. This is why we do not to employ parallelization techniques that are (i) too specific for merge sort or (ii) founded on specific functionality of particular parallel computers.

2.1 Shared Memory Merge Sort with OpenMP

The OpenMP API [13] supports, on a variety of platforms, programming of shared memory multiprocessing. With OpenMP, C/C++ and Fortran programmers use a set of compiler directives (pragmas), library routines, and environment variables to specify multi-threaded execution that is implicitly managed by the OpenMP implementation.

OpenMP supports a straightforward conversion of serial recursive merge sort (Fig. 1) into a multi-threaded recursive merges sort (Fig. 2). A *parallel sections* directive calls for enclosed independent sections of code – as defined by nested instances of the *section* directive – to be divided between automatically generated threads (Fig. 2).

By default, the additional array $temp$ is shared by all threads. Therefore, the second recursive call from the serial version (Fig. 1) must be modified to provide to each thread a unique part of the shared additional $temp$ array (Fig. 2).

```
void mergesort_parallel_omp
(int a[], int size, int temp[], int threads) {
    if ( threads == 1) { mergesort_serial(a, size, temp); }
    else if (threads > 1) {
        #pragma omp parallel sections
        {
            #pragma omp section
            mergesort_parallel_omp(a, size/2, temp, threads/2);
            #pragma omp section
            mergesort_parallel_omp(a + size/2, size - size/2,
                temp + size/2, threads - threads/2);
        }
        merge(a, size, temp);
    } // threads > 1
}
```

Fig. 2. Shared memory parallel merge sort with OpenMP (it uses parallel sections to assign recursive calls to threads)

It is possible to further parallelize the OpenMP merge sort by parallelizing the *merge* operation as well. This can be done by a conversion into OpenMP of a platform-specific technique originally developed for the .Net Task Parallel Library [6]. Reportedly, this technique can make parallel merge sort 25% faster than parallel quick sort, probably because the merge operation is easier to parallelize than quick sort's partition operation.

The performance of the above shared memory (with

OpenMP) implementation has been measured on (i) on a stand-alone multi-core computer and (ii) on an Amazon AWS's large multi-core instance; performance results are reported in Sections 2.4 and 2.5 correspondingly.

2.2 Message-Passing Merge Sort with MPI

The MPI API [12] supports, on a variety of platforms, programming of message-based communication between processes and is typically used in distributed-memory systems, such as computer clusters. With MPI, programmers in a wide variety of languages use a set of library routines to implement communication and synchronization between processes.

Recall that OpenMP threads are dynamically assigned to parallel sections when the execution reaches a parallel section. This means that with OpenMP, the tree of recursive merge sort calls is automatically mapped onto threads. In contrast to OpenMP, all MPI processes start at once at the very beginning of program execution, and all processes concurrently execute the same code – the entire program. Consequently, the MPI program must permit each process to recognize its own place and role in the recursion tree. With MPI, processes need to be explicitly programmed to map themselves to nodes in the recursion tree, while with OpenMP, it is OpenMP itself that straightforwardly maps nodes from the recursion tree to threads. This difference makes the task of the MPI programmer more complicated in comparison to the task of the OpenMP programmer

As MPI processes map themselves to nodes from the recursion tree, they form a virtual *process tree*. Process 0 is at the root of the tree, with the remaining processes appearing as nodes of the tree (Fig. 3). The *root process* splits the data and sends half of it to a *helper process* which sorts the data and returns it to the root process (send operations are visualized as arrows in Fig. 3). The other half of data is retained by the root process for further sorting by using this same procedure (data retention within processes are visualized by dotted lines in Fig. 3). Once sorted, the two halves of data are merged by the root process.

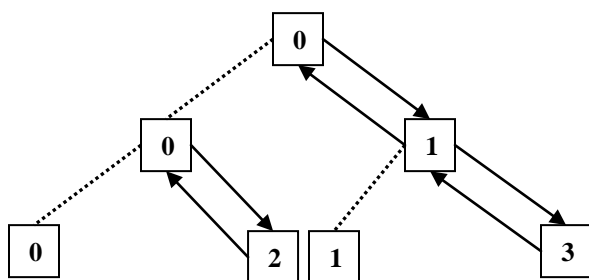


Fig. 3. MPI process tree for recursive merge-sort. Arrows visualize communications with helper processes; dotted lines represent data retained by process for further sorting

Note that the root process can further split its retained data

and send half of it to yet another helper process. Helper processes themselves can follow the same procedure as the root process. Splitting and sending data continues until each MPI process becomes a node in the virtual process tree, i.e. until all processes are sent some amount of data to sort.

All MPI processes run the same main function (Fig. 4) which differentiates between the root process and helper processes. The root process prepares the array to sort and then invokes parallel sort while each helper process: (i) receives data from its parent process; (ii) invokes parallel merge sort; and (iii) sends sorted data back to parent (Fig. 3). Note that each helper process calculates the level of its top-most appearance in the process tree and passes it to the parallel merge sort function (see Fig. 4).

```
int main(...) {
    // ask MPI for my_rank;
    if (my_rank == 0) {
        // allocate array to sort then run root to sort it:
        run_root_mpi(a, size, temp, ...);
    } else {
        run_helper_mpi(my_rank, ...);
    }
    // array is sorted;
}

void run_root_mpi (int a[], int size, int temp[], ...) {
    int level = 0;
    mergesort_parallel_mpi(a, size, temp, level,...);
}

void run_helper_mpi(int my_rank, ...) {
    // probe MPI for a message from parent process
    // and identify message size and parent_rank;
    // allocate int a[size], temp[size];
    MPI_Recv(a, size, ..., parent_rank, ...);
    int level=my_topmost_level(my_rank);
    mergesort_parallel_mpi(a, size, temp, level, ...);
    // send sorted array to parent process:
    MPI_Send(a, size,..., parent_rank, ...);
}

int my_topmost_level_mpi(int my_rank) {
    int level = 0;
    while (pow(2, level) <= my_rank) level++;
    return level;
}
```

Figure 4. Root and helper processes in MPI merge sort

Parallel merge sort is executed by various processes at various levels of the process tree, with the root being at level 0, its children at level 1, and so on (Fig. 3). In that, the process's level and the MPI process rank are used to calculate a corresponding helper process's rank (Fig. 5). Then, merge sort communicates for further sorting half of the array with that helper process. Serial merge sort is invoked when no more MPI processes are available. The helper's rank calculation method is adopted from Perera's MPI quick sort

algorithm [10].

```

void mergesort_parallel_mpi
(int a[], int size, int temp[], int level, ...) {
// my_rank is used to calculate helper rank:
int helper_rank = my_rank + pow(2, level);
if (helper_rank > max_rank) {
    mergesort_serial(a, size, temp);
} else {
// send second half of array, asynchronous:
MPI_Isend(a+size/2, size-size/2, ..., helper_rank, ...);
// sort first half:
mergesort_parallel_mpi(a, size/2, temp, level+1, ...);
// receive second half sorted:
MPI_Recv(a+size/2, size-size/2, ..., helper_rank, ...);
// merge the two sorted sub-arrays:
merge(a, size, temp);
}
}

```

Fig. 5. Message-passing parallel merge sort with MPI. It uses explicit mapping of recursive calls to helper processes

The performance of the above message-passing (with MPI) implementation is evaluated in Section 2.4.

2.3 Hybrid Merge Sort with MPI and OpenMP

A hybrid parallel architecture combines distributed and shared memory in the same computing system. Some authors prefer the term “multi-level” parallel architecture but we choose to use “hybrid” for its brevity. An SMP cluster of multi-processor multi-core nodes is a typical example of a hybrid parallel system. Besides computer clusters, NUMA computers, such as Compaq’s Alpha EV6 and SGI Origin can also be viewed as hybrid parallel systems.

```

void mergesort_parallel_mpi_and_omp
(int a[], int size, int temp[], int level, int threads, ...) {
int helper_rank = my_rank + pow(2, level);
if (helper_rank > max_rank) {
    mergesort_parallel_omp(a, size, temp, threads);
} else {
    MPI_Isend(a+size/2, size-size/2, ..., helper_rank, ...);
    mergesort_parallel_mpi_and_omp
        (a, size/2, temp, level+1, threads, ...);
    MPI_Recv(a+size/2, size-size/2, ..., helper_rank, ...);
    merge(a, size, temp);
}
}

```

Fig. 6. Hybrid parallel merge sort with MPI and OpenMP

Recursive merge sort can be mapped rather straightforwardly onto a hybrid parallel architecture by means of MPI and OpenMP. On a hybrid system, MPI can provide

coarse-grain parallelism by mapping merge sort recursive invocations onto a process tree (Fig. 3), as already discussed in Section 2.2. In addition, OpenMP can provide finer-grain parallelism by introducing multiple threads within individual MPI processes, namely those MPI processes that are visualized as leaf nodes in the process tree (Fig. 3). A more formal outline of this approach is shown in Fig. 6.

Note that hybrid merge sort (Fig. 6) switches to shared memory merge sort (rather than to serial merges sort) when no more MPI helper processes are available, thus utilizing all available processors and cores on each cluster node.

The performance of the above MPI + OpenMP hybrid implementation is evaluated in Section 2.4.

2.4 Performance Evaluation

We measured the performance of our shared memory, message-passing, and hybrid parallel merge sorts on a five-node Rocks 5.2 cluster running OpenMPI 1.3 and OpenMP 3 under GNU/Linux. Each cluster node contained two Intel Xeon quad-core processors running under a 2.80 MHz clock. We executed our merge sorts with randomly generated arrays of 10^7 integer elements. Note that cluster node capacity permitted experiments with arrays consisting of up to 30^7 integer elements. No other applications were active on the cluster during our performance measurements.

Table 1. Performance results on a standalone Rocks cluster (all times are in seconds)

Program	OpenMP Threads	MPI Processes	Nodes Used	Cores Used	Average Rocks Time	Rocks Speedup
Serial			1	1	4.1	1.0
OpenMP	2		1	2	2.4	1.7
	4		1	4	1.6	2.6
	8		1	8	1.3	3.2
MPI		8	1	8	2.9	1.4
		16	2	16	2.2	1.9
		24	3	24	2.1	2.0
		32	4	32	1.9	2.2
		40	5	40	2.0	2.1
Hybrid	8	1	1	8	1.3	3.2
	8	2	2	16	1.5	2.7
	8	3	3	24	1.5	2.7
	8	4	4	32	1.9	2.2
	8	5	5	40	1.8	2.3

Shared memory merge sort (with OpenMP, Section 2.1) was executed on 1, 2, 4, and 8 cores on the master node of the Rocks cluster. Message-passing merge sort (with MPI,

Section 2.2) was executed on 1 to 5 nodes by using all available cores on all nodes for MPI processes. Hybrid memory merge sort (with MPI and OpenMP, Section 2.3) was executed on 1 to 5 nodes by using one core on each node for distributed MPI processes and all 8 cores for shared memory OpenMP processes. Table 1 presents average wall-clock times and speedup for serial, shared memory, message-passing, and hybrid merge sorts.

In our experiments, shared memory merge sort runs faster than message-passing merge sorts. Hybrid merge sort, while still slower than shared memory merge sort, is faster than message-passing merge sort.

Different OpenMP sections (see Fig. 2) may – or may not – be executed by different threads. It is up to the runtime environment to assign threads to sections, and the OpenMP programmer has no control over thread-to-section assignment. Our experiments show that the runtime environment may overuse some threads and underuse others, as illustrated by Table 1. Despite of this inadequate load balancing, shared memory merge sort with OpenMP still performs faster than message-passing merge sort with MPI.

Table 2. Merge sort calls per a thread in a test execution

Thread #	0	1	2	3	4	5	6	7
Assigned calls	4	3	3	0	1	0	1	3

Our merge sorts process a single array that can be entire held in RAM on a single node. This setup is advantageous for single node implementations with OpenMP and disadvantageous for multiple-node implementations with MPI. Indeed, such a centralized setup involves multiple MPI data transmissions that begin and end with the root node; at the same time, OpenMP is exempt from such transmissions. Should the setup change to permit handling of “big data” that do not fit in a single node RAM, all implementations would require multiple I/O operations. In a “big data” setup, MPI’s parallel I/O functionality may possibly provide considerable advantages in comparison to pure OpenMP implementations.

2.5 Parallel Merge Sort on the Amazon Elastic Compute Cloud

Amazon Web Services (AWS) is the first – and currently the largest – public cloud computing platform that provides virtual computing resources on a metered, pay-per-use basis [1]. A goal of the AWS development was to offer as a public utility part of the extensive Amazon data centers by means of service-oriented virtualization. AWS incorporates a number of services, most notably the Elastic Compute Cloud (EC2), and also services built on top of EC2, such as the Elastic MapReduce.

Using AWS’s EC2, we (i) launched a single server instance; (ii) uploaded and compiled our OpenMP-based shared memory merge sort; (iii) ran merge sort experiments

and collected performance data; (iv) terminated the server instance. In the process, we were charged only for the actual time during which our server instance was running, and the charges were covered by a grant provided by Amazon.

To launch our AWS server, we used an abstract machine image (AIM) provided by Amazon itself, a CentOS system with an OpenMP-enabled C-compiler readily available. We launched this AIM as a single 64-bit cluster compute instance with 8 physical cores from two quad-core Intel Xeon processors, running under a 2.93 GHz clock. With hyper threading, the server provided 16 virtual cores. We executed our shared memory merge sort with randomly generated arrays of 10^7 integer elements, much like we did on the standalone Rocks cluster (Section 2.4). Note that AWS server capacity permitted experiments with arrays consisting of up to 10^9 integer elements, much larger than the 30^7 limit of our standalone Rocks cluster nodes. On AWS, we chose to systematically experiment with arrays of 10^7 integer elements for the sake of performance comparisons with the standalone Rocks installation.

Shared memory merge sort (with OpenMP) was executed on 1, 2, 4, 8, and 16 cores on our AWS server. Table 3 presents average wall-clock times and speedup – for serial and shared memory merge sorts on the AWS virtual server. For the sake of more convenient comparisons, Table 3 includes standalone Rocks cluster performance data from Table 1 (Section 2.4).

Table 3. Performance results on an AWS virtual server

Program	OpenMP Threads	Virtual Cores Used	Physical Cores Used	Average AWS Time	Average Rocks Time	AWS Speedup	Rocks Speedup
Serial		1	1	2.5	4.1	1.0	1.0
OpenMP	2	2	2	1.4	2.4	1.8	1.7
	4	4	4	0.8	1.6	3.1	2.6
	8	8	8	0.5	1.3	4.7	3.2
	16	16	8	0.5		4.7	

Time and speedup data from Table 3 clearly indicate that our rented AWS instance, being a physically hosted on a new and more powerful shared memory computer, offered a clear advantage in terms of performance as compared to our dedicated Rocks node.

Looking at Table 3, one may conclude that hyper threading is not particularly beneficial for our shared memory merge sort. In fact, we found out that hyper threading becomes a positive factor for larger arrays. Shared memory merge sort (with OpenMP) was executed on 8 physical and 16 virtual (with hyper threading) cores on our AWS instance. Table 4 outlines performance of serial and shared memory merge sorts on a set of very large arrays. This table clearly indicates a speedup gain from hyper threading for larger arrays.

Table 4. Performance with hyper threading on large data

OpenMP Merge Sort Data Size	Serial Time	8 Physical Cores - Time	16 Virtual Cores - Time	8 Physical Cores - Speedup	16 Virtual Cores - Speedup
10^7	2.5	0.5	0.5	4.7	4.7
10^8	29.5	5.4	4.9	5.4	6.0
$5 \cdot 10^8$	161	28.8	24.4	5.6	6.6
10^9	334	59.5	50.1	5.6	6.7

While launching and using a single high-performance AWS instance is straightforward, configuring a multi-node virtual MPI cluster on AWS is not as easy. As of the time of this writing (March 2011), we are not aware of good quality generic AMIs that can be used to launch MPI clusters by following well documented, sound procedures. At the absence of pre-packaged MPI-enabled cluster nodes, users who would like to run MPI on AWS must act as system administrators and build MPI-enabled, cluster-capable AMIs by themselves. Despite of the technical difficulty of the process, we managed to configure and fire a virtual SMP cluster on the on the Amazon EC2.

To launch our AWS cluster, we created a custom AIM, an Ubuntu Lucid system enhanced with MPI and containing our own merge sort programs. We used this AIM to fire an MPI cluster of five extra-large EC2 instances. In AWS terminology, each instance was a 64-bit platform with 4 virtual cores. Again, we executed our message-passing merge sort with randomly generated arrays of 10^7 integer elements, just like we did on the standalone Rocks cluster (Section 2.4).

Note that although our AWS cluster and the standalone Rocks cluster consisted of the same number of nodes, the two clusters differed in their node architectures, including the number of cores in each node (8 physical cores on the Rock cluster as opposed to 4 virtual cores on the AWS cluster). This is why it is difficult to formally compare performance results obtained on these different clusters with our message-passing merge sort. Yet, it became clear that our message-passing merge sort achieved higher performance on the Rocks cluster than on the AWS cluster. More important, execution times that we measured on the AWS cluster were unstable and varied in a much larger range than execution times on the Rocks cluster.

Data in Table 5 illustrate the performance instability of the AWS virtual cluster. Table 5 includes a representative selection of: average, minimal, and maximal wall-clock times; corresponding standard deviations; corresponding speedup data for message-passing merge sort on the AWS virtual cluster and, for comparison, on the Rocks cluster.

Table 5. Performance deviations, AWS and Rocks clusters

Platform	Nodes	Total Cores	Aver Time	Min Time	Max Time	Standard Deviation	Speedup
AWS	4	16	3.3	2.5	4.9	0.9	1.2
Rocks	4	32	1.91	1.89	1.93	0.02	2.2
AWS	5	20	13.6	3.0	40.5	11.9	0.3

Rocks cluster's performance advantages over the AWS cluster can be attributed to the following factors. First, our AWS virtual nodes, being AWS EC2 instances, shared the same hardware with other unknown AWS instances and their applications, and load spikes in those anonymous applications had been probably quite detrimental to the AWS virtual cluster performance; in contrast, we had the Rocks cluster dedicated to our experiments. Second, our Rocks cluster nodes were physically located on the same rack while our AWS virtual nodes were located in the same region, but quite likely on different racks; thus slower and busier network connections negatively affected AWS cluster performance and stability. Last but not least, virtualization in EC2 may induce significant penalties for scientific computing workloads [14].

3 Conclusions

This paper introduces three parallel versions of recursive merge sort: shared memory (with OpenMP), message-passing (with MPI) and hybrid (with MPI and OpenMP). While others have developed merge sort algorithms with either multi-threading [6] or message-passing [11], this paper offers comparable multi-threaded, message-passing, and hybrid implementations. The paper reports performance experiments with the three approaches and draws conclusions accordingly (while neither [6], nor [11] report systematic performance results of their individual algorithms). Our performance experiments show that shared memory merge sort (with OpenMP) is faster than message-passing merge sort (with MPI) when applied to arrays that fit entirely in RAM; the performance of hybrid merges sort falls between that of shared merges sort and message passing merge sort. (These relations, however, may not hold for very large arrays that significantly exceed RAM capacity.) Note, however that which programming paradigm is best widely depends on the nature of the problem, the hardware and software in cluster nodes, and the cluster network; for example, a fast network can make a message-passing (with MPI) solution for some problem faster than shared-memory (with OpenMP) and hybrid solutions [15].

Last but not least, this paper describes cloud computing experiments with shared memory merge sort (with OpenMP) and with message-passing merge sort (with MPI) on AWS in general and on the Amazon Elastic Compute Cloud in

particular. The use of OpenMP on AWS is straightforward; it has led us to a better speedup, thanks to the readily available high-performance instance on a pay-per-use basis. In contrast, MPI virtual clusters are not readily available on AWS and their configuration for AWS requires technical system administration skills. Our experiments with a virtual AWS cluster exhibited poor and unstable performance. A recent EC2 benchmark performance analysis concludes that the performance and reliability of the EC2 cloud are low [14]. Yet, the EC2 cloud “may still appeal to scientists who need resources immediately and temporarily” [14]; our shared memory merge sort EC2 experiments demonstrate that specific problems and software may actually give performance gains to the high-performance cloud user.

4 References

- [1] Amazon Web Services. Retrieved on March 1, 2011 from <http://aws.amazon.com/>.
- [2] Cole , Richard. Parallel merge sort. *SIAM Journal on Computing*, Volume 17 Issue 4, August 1988, 770-785.
- [3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. *Introduction to Algorithms* (3rd ed.). MIT Press, 2009.
- [4] El-Ghazawi, Tarek; Carlson, William; Sterling, Thomas; Yelick, Katherine. *UPC: Distributed Shared Memory Programming*. Wiley, 2005.
- [5] Geist, Al; Beguelin, Adam; Dongarra, Jack; Jiang, Weicheng; Manchek, Robert; Sunderam , Vaidy. *PVM: Parallel Virtual Machine*. MIT Press, 1994.
- [6] Huba , Dzmitry. Parallel merge sort. Retrieved on March 1, 2011 from <http://dzmitryhuba.blogspot.com/2010/10/parallel-merge-sort.html>.
- [7] Katajainen, Jyrki; Träff, Jesper L. A meticulous analysis of mergesort programs. *Lecture Notes in Computer Science*, 1997, Volume 1203/1997, 217-228.
- [8] LaMarca, Anthony; Ladner, Richard. The influence of caches on the performance of sorting. *Proc. 8th Ann. ACM-SIAM Symposium on Discrete Algorithms (SODA97)*, 370-379.
- [9] Orio: An Annotation-Based Empirical Performance Tuning Framework. Retrieved on March 1, 2011 from <http://trac.mcs.anl.gov/projects/performance/wiki/Orio>.
- [10] Perera, Prasad. Parallel quicksort using MPI & performance analysis. Retrieved on March 1, 2011 from http://www.codeproject.com/KB/threads/Parallel_Quicksort/Parallel_Quick_sort_without_merge.pdf.
- [11] Rolfe ,Timothy J. A Specimen of parallel programming: Parallel merge sort implementation. *ACM Inroads*, Volume 1, Issue 4, December 2010, 72-79.
- [12] The Message Passing Interface (MPI) standard. Retrieved on March 1, 2011 from <http://www.mcs.anl.gov/research/projects/mpl/>.
- [13] The OpenMP specification for parallel programming. Retrieved on March 1, 2011 from <http://openmp.org>.
- [14] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. A performance analysis of EC2 cloud computing services for scientific computing. In: *Cloud Computing: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer, 2010, Vol. 34, Book Series Editor: O. Akan et al., pp. 115-131.
- [15] G. Jost, H. Jin, D. Mey, F. Hatay. Comparing the OpenMP, MPI, and hybrid programming paradigms on an SMP cluster. *NAS Technical Report NAS-03-019*, November 2003.

Rapid Performance of a Generalized Distance Calculation

Scott Fisackerly, Eric Chu, David L. Foster

Electrical and Computer Engineering Department, Kettering University, Flint, MI, USA

Abstract- *The ever-increasing size of data sets and the need for real-time processing drives the need for high speed analysis. Since traditional CPUs are designed to execute a small number of sequential process, they are ill-suited to keep pace with this growth and exploit the massive parallelism inherent in these problem spaces. In the last several years, the parallelism of GPUs has made them a viable solution for general purpose computing. However, effective use of GPUs requires a significantly different programming paradigm. Towards the goal of creating a function library that maximizes the performance improvement of GPUs in data analysis and clustering, this paper presents an implementation of a general n-dimensional distance calculation commonly used in these types of algorithms. Experimental results show up to a 390x speedup using a Tesla C1060 and up to a 538x speedup using a GeForce GTX 480 over an Intel Core i7.*

Keywords: GPU Computing, Distance Calculation, Parallel Computing

1 Introduction

The magnitude of large data sets and input streams are growing rapidly. Examples can be seen in the recent digitalization of the medical database in America, the amount of experimental data that the Large Hadron Collider at CERN can produce, and the rising number of sensors in vehicles, manufacturing processes, and security systems. This flood of data heralds a critical need for data analysis and clustering algorithms that can make large-scale computing faster and more affordable and increase the complexity and ability of real-time systems.

In the last several, GPU computing has become a viable alternative for general-purpose computing. For comparable throughput, a GPU can often be purchased at one tenth of the cost of traditional CPU, and they can be operated at one twentieth the electricity [1]. GPUs are also available for mobile devices, making them an option in ubiquitous systems. However, the full advantages of GPUs for general purpose computing cannot be realized by merely existing algorithms to compile and generate the correct answer. Existing algorithms must be recast correctly to efficiently leverage the GPU platform, and there remains a tremendous amount of work in this area

A library of GPU-accelerated functions is being developed to facilitate several different data analysis and clustering research projects for both large-scale and real-time computing. This paper focuses on a generalized distance calculation that operates on vectors with n

dimensions of floating points numbers. As will be discussed in Section III, this function's flexibility allows it to calculate Manhattan distances, Euclidean distances, and other variations. It is specifically targeted to several forms of fast fuzzy clustering for extremely large data sets and real-time fuzzy clustering. Several full clustering algorithms have been implemented with reported speedups. One method of K-means clustering focuses on data sets that exceed the GPU memory size, with reported speedups of about 10x over a highly optimized, 8-core CPU implementation [2]. Hierarchical clustering algorithms have achieved a 65x speedup [3]. Other K-Means implementations have gained a 40x speedup [4]. However, there is little discussion in the literature about the individual functions and their respective speedups, the bottlenecks of existing algorithms, and in many cases the specifics of the GPU implementation. This work details a specific, flexible function with a significant speedup that can be incorporated into various algorithms. Also, we report the optimizations that lead to the speedup so that the concepts may potentially be applied to other functions.

The remainder of the paper is outlined as follows. Section II gives a general background of GPU computing along with the issues that much be addressed to efficiently port an algorithm to this platform. Section III discusses the generalized distance calculation explored in this work. Section IV describes the optimizations applied to the GPU version of the algorithm, and Section V relates the experimental results. Section VI concludes the paper.

2 GPU Computing Background

There are three main GPU computing platforms currently available: NVIDIA's CUDA (Compute Unified Device Architecture), OpenCL, and DirectCompute. This work uses CUDA for several reasons. It is a relatively straightforward extension to the C language. It has significant market penetration, reportedly used by over 60,000 researchers and in over 400 financial institutions. Also, NVIDIA estimates that over 250,000,000 CUDA capable video cards have been shipped by 2010 [5].

As stated, CUDA is an set of extensions for several programming languages, notably C. It provides services to create data structures in the GPU memory, transfer data between GPU memory and CPU memory, call kernel functions on the GPU, and other utilities. CUDA capable

devices can be queried in software for their major and minor hardware revisions, allowing programs to select more optimal parameters for the specific GPU. However, since the GPU processing unit is largely abstracted, significant software development can be completed without regard to the end hardware. More details can be found in [6, 7].

GPU computing follows a significantly different paradigm than traditional CPU computing. With CUDA, there are three main aspects that must be addressed for efficient and fast execution. The first main aspect is decomposing the problem space and mapping it into CUDA's virtual representation so that a kernel can be scheduled on the GPU's processing units. A processing unit is referred to as a streaming multiprocessor (SM) and contains an array of streaming processors (SP). Early cards used 8 SP's, each handling four threads simultaneously, and recent 400 series and 500 series GPUs use 32 SPs handling one thread each. A SM operates on a group of 32 threads, called a warp, to execute 32 threads simultaneously. One of the primary differences between GPUs is the number of SMs available. Smaller GPUs intended for laptops or basic graphics may have as few as 2 SMs, while GPUs intended for intense computing, such as that in the Tesla C1060, may have up to 30 SMs per chip.

CUDA threads are organized into three-dimensional blocks, with each thread having a unique x, y, and z coordinate within the block. Blocks are in turn arranged into a three-dimensional grid, with each block having a unique x, y, and z coordinate. These indices can be accessed by threads through special variables during execution. Thus, each thread can calculate a unique thread index used for accessing data structures. There are limitations on the range of indices in each dimension for threads and blocks as well as an upper limit on the total number of threads per block. These limitations are specific to the hardware revision and can be queried in software. As a rule of thumb, blocks should contain a multiple of 32 threads so that it contains an even number of warps. Common values are 128 and 256 threads per block. The GPU schedules blocks to execute on SMs, and if an SM has sufficient resources, multiple blocks may be scheduled on the same SM.

When mapping this virtual organization onto a data structure, it is important to exploit enough parallelism to generate a large number of threads. While CPUs mitigate memory access latency with a cache structure, GPUs hide this latency by switching warps. Current SMs have enough registers to hold 24 or 32 warps simultaneously, and the SM can quickly switch between them performing a context switch without the need to save status in memory. Without a sufficient number of threads to select from, the SMs must idle while waiting on memory accesses. Proper block and grid selection can also affect the following major aspect.

Coalesced memory accesses also has a major influence on GPU performance. GPU addresses point into small blocks of memory at a specific memory pitch, such as every 64 of 128 bytes. Memory accesses are performed by half-warps,

with 16 threads performing their reads or writes concurrently. If these 16 reads are on contiguous RAM addresses, the access can complete in 1 read if the access doesn't cross an address boundary, or two accesses if a boundary is crossed. If these accesses are spread throughout the RAM, the half-warp will take 16 memory accesses. With memory accesses requiring about 400 to 600 clock cycles, careful design of the kernel, data-structures, and access patterns can significantly reduce the memory latency.

The final main aspect of efficient CUDA kernel design is appropriate use of the available memory types. Each thread has access to a set number of registers, as limited by the hardware revision. Naturally, register access is the fastest available. Each block has access to a limited amount of shared memory located in the SM. A SM has 8 KB to 64 KB depending on the hardware revision. All threads in a block can share data in the block's portion of shared, but this memory must be divided among all blocks scheduled concurrently on the SM, and blocks cannot read each other's shared memory. The amount of shared memory required per block is also a limited factor on how many blocks can be simultaneously scheduled on the SM. Shared memory is relatively fast, on the order of 10 clock cycles, and is often used as programmer-managed cache. The card has constant memory and texture memory visible to all threads in the kernel. The slowest memory is the GPU's RAM, often 400-600 cycles. Global variables reside in RAM. Of major importance, if the number of registers is insufficient for a thread's local variables, excess local variables are allocated in RAM reducing the performance.

3 Function Implementation

This section describes the general function that was implemented, summarizes the implementation itself, describes the mapping scheme, and discusses optimization strategies.

3.1 Functional Description

As noted previously, a common operation in many clustering algorithms is a distance calculation that generates a measure of dissimilarity of two vectors. The implemented formula is shown below in Equation 1.

$$distance(\mathbf{p}, \mathbf{q}) = \sqrt[s]{\sum_{i=1}^n (|p_i - q_i|)^s} \quad (1)$$

This formula returns the distance between two vectors, p and q , that each have n dimensions of floating-point data. For two common distance metrics, if $s=1$, this function calculates the Manhattan distance, and if $s=2$, it calculates the Euclidean distance.

3.2 Implementation

The CUDA function was designed to handle large arrays of vectors. It requires two input matrices. Matrix P is p by n points, representing p vectors of n dimensions. Similarly, matrix Q is q by n points representing q vectors of n points. The kernel is also passed a pointer to matrix D, which is a p by q matrix used to return the resulting distances in which element (i,j) is the distance between the ith vector in P and the jth vector in Q. This format is useful so that the function can calculate the distances between p input vectors to q established data clusters, or the same matrix can of course be passed as both P and Q to determine the distances between a set of vectors. The function also requires p, q, and n as integer-valued parameters, and s as a floating-point input. Unlike some "fast" implementations for functions in CUDA, array bounds are checked within the kernel code, and there are no resulting limitations on p, q, and n.

3.3 Grid and Block Mapping

To map the problem into CUDA's thread organization, the grid size was set to a one-dimensional array of p blocks. Each block would calculate the distance between one of the p vectors in array P and all q vectors in the Q array. The block size was set to a one-dimensional array of 256 threads. Each thread would calculate the distance between the block's assigned input vector from P and 1/256 of the q vectors from Q.

3.4 Function Optimizations

Several implementations of the GPU kernel were developed. The first was a baseline version, GPU-Baseline, concerned only with generating the correct results and without any specific effort for optimization. It was noticed when analyzing the performance of this version that, since C stores arrays row-wise in memory, the Q array was being accessed in a pathological access pattern with every half-warp access being broken into 16 separate reads. To alleviate this issue, the Q array was transposed first using sample code from the CUDA SDK [8], and then the distance calculation was performed, yielding coalesced memory accesses. This version is referred to as GPU-Transpose.

The second observation was that the P matrix was being accessed one element at a time. All threads in a block were loading the ith element from a vector in P and then accessing all of the ith elements in the vectors in Q. This required the kernel to access the P array in RAM once for every dimension during the kernel. This pattern was improved by using shared memory in the SM's. The kernel uses all threads to fetch 256 values from a vector in P and store them in shared memory. The next 255 loop iterations would fetch the values from shared memory instead of RAM, drastically lowering the latency. This version is referred to as GPU-Shared, and it should be noted that this version also uses the transpose optimization discussed previously.

4 Testing and Results

The computers used for testing contained the following: an Intel® Core™ i7-920 processor at 2.66 GHz with 8 MB of L3 cache, 6GB of PC10666 RAM, an ASUS P6T Deluxe motherboard, an EVGA 260 GTX for graphics, and either an NVIDIA Tesla C1060 or a Fermi-based EVGA GeForce GTX 480 for GPU computing. The operating system was Windows 7, and the system used CUDA Toolkit 3.2. Details of the C1060 used for GPU computing can be found in NVIDIA's data sheet [9].

Several sets of values were tested for n, p, and q. Figure 1 shows a subset of values used in testing. For each test, values for input arrays P and Q were generated randomly, and this time was not included in the measurements. The known correct version of single-threaded CPU code, referred to as CPU-Baseline, was executed on the input set and the calculation time was recorded. Then, the GPU code was executed on the same data and was compared to the CPU results to ensure correct operation. For test cases that had very short running times on the GPU (some had less than a millisecond), each set of parameters was used for 100 tests, and the running times were accumulated, then divided by 100. It should be noted that the times measured for the GPU included data transfers to and from the card, but they did not include the validation with the CPU results. Each of these tests was then run completely 5 times.

	n	p	q
TPV1	1024	32768	1024
TPV2	512	16384	512
TPV3	256	8192	256
TPV4	256	16384	256
TPV5	256	32768	256
TPV6	256	8192	512
TPV7	256	8192	1024

Figure 1 Subset of Test Parameter Values

The running times of the functions were expected to be linearly proportional to n, p, and q and invariant to s, and these relationships were confirmed by experimental results. The GPU versions showed slightly more variation due to memory transfer operations between the CPU and GPU and block scheduling. Figure 2 and Figure 3 show the average running times for a subset of results that demonstrates these patterns. The standard deviation in running times for all versions was extremely small, less than 0.1%. Full results are not shown for space considerations and since the results correlate strongly with the above patterns.

	CPU-Baseline	GPU-Baseline	GPU-Transpose	GPU-Shared
TPV1	5466.232	39.082	21.956	14.158
TPV2	684.012	4.885	2.815	0.951
TPV3	85.557	0.775	0.375	0.253
TPV4	171.138	1.522	0.722	0.510
TPV5	342.729	2.685	1.448	0.953
TPV6	171.227	1.333	0.754	0.505
TPV7	342.741	1.769	1.446	0.956

Figure 2 Average running times in seconds on Tesla C1060

	CPU-Baseline	GPU-Baseline	GPU-Transpose	GPU-Shared
TPV1	5466.232	20.444	14.429	10.158
TPV2	684.012	2.235	1.831	1.370
TPV3	85.557	0.292	0.224	0.252
TPV4	171.138	0.587	0.478	0.418
TPV5	342.729	1.150	0.946	0.751
TPV6	171.227	0.560	0.474	0.340
TPV7	342.741	1.152	0.931	0.662

Figure 3 Average running times in seconds on GTX 480

The experiments demonstrated that the transpose optimizations and the shared memory optimization greatly improved the performance of the GPU code on the Tesla, and that the speedup of the best GPU version was substantially better than the CPU code, as shown in Figure 4 and Figure 5. The transpose operation improved performance by 1.18 to 2.27 times and 1.17 to 1.42 times for the Tesla and GTX 480 respectively, and use of shared memory increased performance by an additional 1.41 to 1.56 times and 1.13 to 1.42 times for the two architectures. It is likely that the smaller amount of improvements with the GTX 480 over the C1060 is because it employs cached memory, and this card would typically have more efficient memory accesses than the Tesla C1060. While the CPU version can be improved by making it multi-threaded, this would produce only a theoretical 8x improvement on the Core i7, and with standard overhead, the realized speedup would be a little less. Even with an 8x speedup in the CPU versions, the GPU versions show significant improvement. The main contributor to execution time was the power function required by the equation. The functions were modified so that $s = 1$ would be a special case, and the kernels would execute without using the power functions. They are referred to as CPU-Manhat and GPU-Manhat, based on CPU-Baseline and GPU-Shared respectively. Figure 6 and Figure 7 show the running times on the two GPUs tested, resulting in a 181 to 422 times speedup and a 434 to 755 times speedup in the Tesla C1060 and GTC 480

respectively. With the small amount of computation in the Manhattan versions, the memory latency and transpose operations were a significant portion of the total running time.

	GPU-Baseline	GPU-Transpose	GPU-Shared
TPV1	139.86	248.96	386.08
TPV2	140.01	242.97	360.55
TPV3	110.37	227.89	337.93
TPV4	112.43	236.99	335.29
TPV5	127.63	236.72	359.63
TPV6	128.44	227.03	338.64
TPV7	193.71	237.06	358.46

Figure 4 Speedup relative to CPU-Baseline with Tesla C1060

	GPU-Baseline	GPU-Transpose	GPU-Shared
TPV1	267.38	378.83	538.12
TPV2	294.11	373.55	527.37
TPV3	292.48	349.21	336.60
TPV4	291.70	357.30	409.23
TPV5	298.09	362.18	456.12
TPV6	305.87	361.24	503.91
TPV7	297.61	368.15	517.55

Figure 5 Speedup relative to CPU-Baseline with GTX 480

	CPU-Baseline	CPU-Manhat	GPU-Shared	GPU-Manhat
TPV1	5466.232	1059.611	14.158	2.510
TPV2	684.012	132.566	0.951	0.417
TPV3	85.557	16.583	0.253	0.091
TPV4	171.138	33.147	0.510	0.121
TPV5	342.729	66.278	0.953	0.181
TPV6	171.227	33.157	0.505	0.152
TPV7	342.741	66.228	0.956	0.199

Figure 6 Average running times in seconds on C1060

	CPU-Baseline	CPU-Manhat	GPU-Shared	GPU-Manhat
TPV1	5466.232	1059.611	10.158	1.404
TPV2	684.012	132.566	1.370	0.201
TPV3	85.557	16.583	0.252	0.038
TPV4	171.138	33.147	0.418	0.071
TPV5	342.729	66.278	0.751	0.130
TPV6	171.227	33.157	0.340	0.065
TPV7	342.741	66.228	0.662	0.112

Figure 7 Average running times in seconds on GTX-480

5 Conclusions

This paper demonstrates tremendous speedups of a general distance calculation function that can be incorporated into many various classification and analysis algorithms. Additionally, it shows that preprocessing is a useful stop to allow more advantageous memory access patterns on the graphics card. It also shows that this particular GPU is significantly more efficient when calculating power functions compared to CPUs, and this could be a determining factor when selecting the most appropriate platform for other functions. With the optimizations given, this work boasts up to an astonishing 538 times speed improvement versus a traditional CPU for the discussed distance calculation. For future work, an optional covariance matrix will be added so the functionality can be extended to Mahalanobis calculations.

6 Acknowledgments

This research was supported by an equipment donation from the NVIDIA Corporation as part of the Academic Partnership Program.

References

- [1] NVIDIA. (2011, Jan 27, 2011). *High-Performance Computing - Supercomputing with Tesla GPUs*. Available: http://www.nvidia.com/object/tesla_computing_solutions.html
- [2] R. Wu, *et al.*, "Clustering billions of data points using GPUs," presented at the Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop, Ischia, Italy, 2009.
- [3] S. A. A. Shalom, *et al.*, "Hierarchical Agglomerative Clustering Using Graphics Processor with Compute Unified Device Architecture," presented at the Signal Processing Systems, International Conference on, 2009.

[4] B. Hong-tao, *et al.*, "K-Means on Commodity GPUs with CUDA," presented at the Computer Science and Information Engineering, World Congress on, 2009

[5] NVIDIA. (2011, Jan. 30, 2011). *What is CUDA?* Available: http://www.nvidia.com/object/what_is_cuda_new.html

[6] J. Sanders and E. Kandrot, *CUDA by Example, An Introduction to General-Purpose GPU Programming*: Addison Wesley, 2010.

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

[8] G. Ruetsch and P. Micikevicius. (2009, Jun 11, 2010). *Optimizing Matrix Transpose in CUDA*. Available: <http://developer.download.nvidia.com/compute/cuda/sdk/web-site/C/src/transposeNew/doc/MatrixTranspose.pdf>

[9] NVIDIA. (2009, Jun 11, 2010). *Tesla C1060 Computing Processor Board*. Available: http://www.nvidia.com/docs/IO/43395/BD-04111-001_v05.pdf

GPU Cluster with MATLAB

A. Guillén M. García L. J. Herrera H. Pomares I. Rojas

Abstract—This paper presents the architecture of an heterogeneous cluster where each node has one or more Graphical Unit Processors (GPUs). The motivation of the work is the fact that this technology presents very impressive results in High Performance Computing at a very low cost and very small energy consumption so. Although this might not be a huge novelty, it is the fact that it can be programmed using MATLAB (one of the scientist's favourite programs). As an example of application, an implementation of the k -Nearest Neighbours will be executed on the platform using both parallelism techniques: MPI and CUDA.

I. INTRODUCTION

As machine learning evolves and more real world applications are faced, more computational power is needed due to the complexity of the problems tackled and the large size datasets. Scientist, aware of this bounds, have been doing research on how to obtain better performance taking advantage of the current technology. The mother nature shows us some examples where "the union makes the force" allowing this union to "divide and conquer" problems. The "ensembling" approach has been analyzed theoretically in models, showing better performances [1]. Computer architecture has squeezed this approach since its very begging ending up in the Internet as a resource provider (the cloud). This paper presents an easy way to use the most recent High Performance Computing (HPC) devices, such as Graphical Processing Units (GPUs), in combination with the classic cluster/beowulf [2] approach.

The rest of the paper is organised as follows: Section 2 will describe some technical aspects in order to understand the architecture, which is described in Section 3. Afterwards, Section 4 will present an example of a parallel application running in the platform developed. Finally, conclusions will be discussed in Section 6.

II. TECHNICAL DESCRIPTION

A. Cluster of computers

Clusters of computers have been used since the early stages of the computer science as a way to distribute work, data and obtain more security and fault tolerance. In 1994, the Beowulf project was developed using a set of desktop personal computers and connecting them through an Ethernet local network.

1) *Programming paradigm: Message Passing Interface (MPI)*: As it is defined in <http://www-unix.mcs.anl.gov/mpl/>, MPI is:

...a library specification for message-passing, proposed as a standard by a broadly based

committee of vendors, implementers, and users..."

Among the advantages of, MPI that have made this library well known, are: that is freely available, was designed for high performance on both massively parallel machines and on workstation clusters, etc.

The Message Passing Interface was designed in order to provide a programming library for inter-process communication in computer networks, which could be formed by heterogeneous computers. The processes communicate among each others by sending messages between pair of processes or between collective communications among groups of processes.

MPI is the most used library for inter-communication in High-performance computing (HPC) application. There are several vendors and public implementations availables Open-MPI¹ and MPICH², for instance. It allows the programmers to use several processes that can be executed in distributed machines.

This library can be used to program all types of HPC computers and is compatible with the Sun Grid Engine (SGE) making the programming and experimentation very comfortable for the users. The library is available in many languages such as C, C++, Java, .NET, python, Ocaml. A special effort was made to be able to use it in MATLAB since Mathworks doesn't provide the full functionality of the library. Regarding the adaptations and interfaces, there are popular ones like MatlabMPI [3], MPITB [4][5]³ and MPI mex [6].

B. Clusters of GPUs

The definition of cluster of GPU is not clear because it might have two interpretations: a machine with several GPUs (that could have several cores), a collection of computers, each one of them with one GPU.

The first approach has been used by M. van Heeswijk et al [7] using the NVidia GTX295 with 2 Graphics Processing Units (GPUs) that provided a total of 1790 GFlops of computational power to design neural networks. In [8] this architecture is exploited as well to improve performance.

The second interpretation of cluster of GPUs has been implemented in more works in the literature. For example by Fan et al [9] built a cluster with 32 computation nodes connected by a 1 Gigabit Ethernet switch. Each node consists of a dual-CPU HP PC with an nVIDIA GeForce FX 5800 Ultra.

Other examples that includes GPUs and CPUs for computing is for example the Lincoln [10] cluster with 192 Servers

¹<http://www.open-mpi.org/>

²<http://www-unix.mcs.anl.gov/mpl/mpich1/>

³This project has been held in MATLAB and continued for Octave

nodes with 96 NVIDIA Tesla S1070 Accelerator Units developed in National Center for Supercomputing Applications (NCSA) located at University of Illinois. Lincoln cluster is an heterogeneous one because it uses CPU and GPU depending on the application of the computation it executes.

1) *Programming paradigm: (Compute Unified Device Architecture) CUDA*: As defined by the company NVIDIA in [11]: "...CUDA is NVIDIAs parallel computing architecture that enables dramatic increases in computing performance by harnessing the power of the GPU (graphics processing unit). ...".

This technology has been well accepted in the research community due to its obvious benefits and possibilities in a wide variety of disciplines (Government & Defense, Molecular Dynamics, Computational Chemistry, Life Sciences, Bioinformatics, Electrodynamics and electromagnetic, Medical Imaging, Financial computing and options pricing, etc. [12]).

As GPUs were becoming more common, some individuals efforts were done [13] in order to call CUDA routines from MATLAB and use these devices, furthermore, some of those products were commercialized [14]. However, all these approaches have become, in a way, obsolete since the MATLAB 2010b Release includes its own Parallel Computing Toolbox which is able to use and manipulate several GPUs [15], [16].

III. PROPOSED ARCHITECTURE: CLUSTER OF GPU CLUSTERS

As describe before, there exists the possibility of setting a cluster using MPI [17] and a cluster of several GPUs [10], [9], furthermore, there exists the possibility of configuring a cluster where each node has several GPUs. For example the BALE cluster at the Ohio Supercomputer Center which has 16 nodes with two dual-core AMD Opteron 2218 CPUs and two Nvidia Quadro FX 5600 GPUs each (that is 2*128 cores). However, as far as we know, nobody up to the date has configured a cluster where each node has several GPUs which can be programmed using MATLAB. Thus, the main contribution of this paper is the presentation of an architecture (Figure III composed by several nodes with several GPUs which can run a MATLAB program. The restriction is that the application has to be deployed using the MATLAB Compiler. Another remarkable aspect of the proposed architecture is that it is made by heterogeneous computers and CPUs not like the previous ones where the cores of the GPUs and the CPUs were, respectively, homogeneous.

A. MATLAB Compiler

MATLAB software has available a tool called *Compiler* which allows MATLAB to generate executable applications (stand-alones) that can be run independently of MATLAB, this is, there is no need of having MATLAB installed in the computer to run the application. The stand-alone requires a set of libraries which can be distributed after being generated with MATLAB, this libraries start the *Component Runtime (MCR)* that interprets the .m files as the MATLAB application would do.

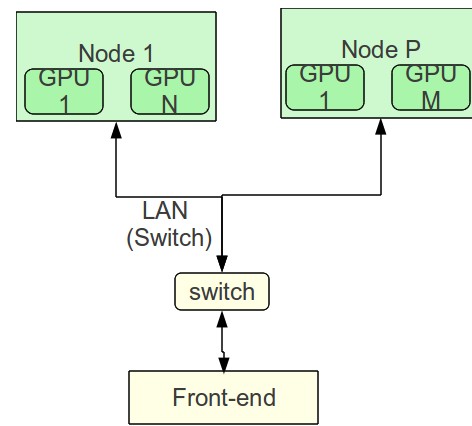


Fig. 1. GPU cluster architecture

The application is packed in an executable file that, when it is run, extracts in a directory all the encrypted .m files that form the deployed application after being compressed so there is no way to access the original source code.

The process that MATLAB follows to generate a stand-alone application is made automatically and totally transparent to the user so he only has to specify the .m files that compose the application to be deployed and MATLAB will perform the following operations:

- Dependence analysis between the .m files
- Code generation: the C or C++ code interface is generated in this step.
- File creation: once the dependencies are solved, the .m files are encrypted and compressed.
- Compilation: the source code of the interface files is compiled.
- Link: the object code is linked with the required MATLAB libraries.

B. Programming paradigm: CUDA + MPI

Using the information from the previous section, the procedure to set up a cluster of GPU clusters⁴ and programming it with MATLAB is straight forward, however, it is not. The implementation of the library MPI might affect the performance, specially if the cluster is using a grid engine to manage the jobs. As far as we know, MPIex [6] has been tested in local clusters and grids configurations where the jobs are sent straight to the nodes having a correct behaviour, therefore, it is the one it will be used in this work. Another reason is that it is based in the OpenMPI implementation which is an active project that is been used in many research labs. This toolbox requires the deployment of the application in order to be sent to the different nodes in the grid and, unfortunately, the Parallel Distributed Toolbox that Mathworks provides does not work on deployed applications, therefore, the support that MATLAB provides to use several GPUs in the same machine cannot be used.

In order to solve this problem, a direct interface to the CUDA routines has to be embedded in the MATLAB code in

⁴considering a GPU cluster as one or more computers with one or more than one GPU interconnected through a Local Area Network

a similar way as it was done with the MPI function calls for MPIex. The embedded routines can be isolated in a toolbox file or written directly in the CUDA code of the soucer .cu files. In both cases, an explicit parameter that selects the GPU to be used must be specified. This parameter should be the rank of a process obtained after the MPIex('Init') call so each process uses its own GPU. The code to do this must be: `GPUselec = (int)mxGetScalar(prhs[N]);` where `(int)mxGetScalar(prhs[N])` makes reference to the parameter number N in the MATLAB function call that should be the process rank. After obtaining this parameter, the CUDA function call to select the GPU is in figure III-B.

To compile the .cu file the following command should be used:

```
./nvmex -f ./nvopts.sh source-code.cu
-I$CUDAROOT/include
-L$CUDAROOT/lib64 -lcudart -lcublas
-lcufft
-L/usr/lib/nvidia-current -lcuda
and defining the variable:
```

```
export MATLAB=$MATLABROOT
```

to indicate the directory where the MATLAB release is installed. The release has to be 2009a since the latests do not support CUDA mex files.

IV. EXPERIMENTS

In this section we will describe an example of a concrete architecture running a concrete code. The code to be run is the computation of the k -Nearest Neighbours which was implemented in CUDA by v. Garcia in [18] and is a quite useful function since this algorithm is widely used in the machine learning field as classifier and regression model.

A. Cluster Architecture

The Grid that was configured had the components described below that were interconnected as Figure IV-A shows.

1) 1 Master node with 2 GPUs:

Processor:

- *model name* : (26) Intel(R) Core(TM) i7 CPU 930 @ 2.80GHz *cache size* : 8192 KB

2 GPUs:

- *Graphics Processor*: GeForce GTS 450
- *CUDA Cores*: 192
- *Memory*: 1024 MB - *Memory Interface*: 128-bit
- *Bus Type*: PCIeExpress x16 Gen1 - *PCI-E Max Link Speed*: 2500

2) 2 Local network nodes with 1 GPU each:

Processor:

- *model name* : (23) Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz *cache size* : 6144 KB

GPU:

- *Graphics Processor*: GeForce 9800 GTX — *CUDA Cores*: 128
- *Memory*: 512 MB - *Memory Interface*: 256-bit
- *Bus Type*: PCIeExpress x16 Gen2 - *PCI-E Max Link Speed*: 5000

Processor:

- *model name* : (15) Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz *cache size* : 4096 KB

GPU:

- *Graphics Processor*: GeForce 8400 GS —*CUDA Cores*: 162
- *Memory*: 512 MB - *Memory Interface*: 64-bit
- *Bus Type*: PCIeExpress x16 - *PCI-E Max Link Speed*: not available

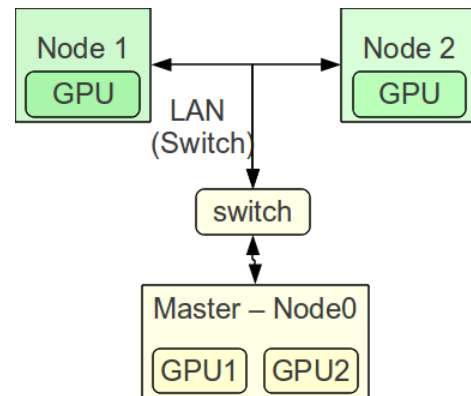


Fig. 5. Cluster Architecture used in the experiments

B. Example of use

In figure IV there is a fragment⁵ of an example code.

The code provided by V. Garcia was adapted and compiled following the indications of the previous section resulting in the following mex file:

```
knn_cuda_with_indexes.mexa64
```

who is invoqued from MATLAB with:

```
[value, indx] =
knn_cuda_with_indexes(ref, query, k,
GPUDevice);
```

where `ref` is the reference set, `query`, is the set that will be consulted, `k`, the number of nearest neighbours to be computed and `GPUDevice` the number of GPU to be used (starting from 0).

The processes divide the reference data set to compute the nearest neighbours of each sub-data set. Afterwards, each process sends its local set of closest neighbours to the root node which performs the final computation with all the subsets computed by the processes

V. CONCLUSIONS

As the time goes by, the Graphic Processing Units are consolidating its position in the High Performance Computing arena. This paper has presented a new framework in order to take advantage of this technology and integrate it with the classical cluster concept. The result is an heterogeneous cluster combining nodes with several GPUs. Another contribution of

⁵for the complete code refer to <http://atc.ugr.es/aguillen/GPUcluster/sampleKNN>.

```

result = cudaSetDevice(GPUselec);
if (result){ printErrorMessage(result); cudaThreadExit(); return; }
result = cudaGetDevice(&cuDevice);
if (cuDevice!=GPUselec) printErrorMessage("Requested GPU couldn't be selected", 0);
if (result){ printErrorMessage(result); cudaThreadExit(); return; }
if(CUDA_SUCCESS!=cuInit(0)) printfErrorMessage("CUDA Initialization failed");

```

Fig. 2. CUDA code that should be run by each process to select the GPU device.

the paper is the description of the procedure in order to run MATLAB programs on the architecture and, as an example, a distributed computation of the k -Nearest Neighbours algorithm has been shown.

Acknowledgments. This research has been supported by the projects by the Spanish CICYT Project TIN2007-60587 and TEC2008-04920 and Junta Andaluca Projects P07-TIC-02768, P08-TIC-03674, P07-TIC-03044, P08-TIC-03903 and P08-TIC03928 and PYR-2010-17 of CEI BioTIC GENIL (CEB09-0010) of the MICINN.

REFERENCES

- [1] A. P. Topchy, M. H. C. Law, A. K. Jain, and A. L. Fred, "Analysis of consensus partition in cluster ensemble," in *Proceedings of the Fourth IEEE International Conference on Data Mining*, ser. ICDM '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 225–232. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1032649.1033458>
- [2] W. Gropp, E. Lusk, and T. Sterling, *Beowulf cluster computing with Linux*. Cambridge, MA, USA: MIT Press, 2002.
- [3] J. Kepner, "Matlabmpi," *J. Parallel Distrib. Comput.*, vol. 64, pp. 997–1005, August 2004. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1032106.1032114>
- [4] J. Fernandez, M. Anguita, E. Ros, and J. Bernier, "SCE Toolboxes for the development of high-level parallel applications," *Lecture Notes in Computer Science*, vol. 3992, pp. 518–525, 2006.
- [5] J. F. Baldomero, "Mpi toolbox for matlab," Universidad de Granada, Tech. Rep., 2005. [Online]. Available: http://atc.ugr.es/javier-bin/mpith/_eng
- [6] A. Guillen, I. Rojas, G. Rubio, H. Pomares, L. Herrera, and J. Gonzalez, "A new interface for mpi in matlab and its application over a genetic algorithm," in *Proceedings of the European Symposium on Time Series Prediction*, 2008, pp. 37–46.
- [7] M. van Heeswijk, Y. Miche, E. Oja, and A. Lendasse, "GPU-accelerated and parallelized ELM ensembles for large-scale regression," *Neurocomputing*, 2010, to appear.
- [8] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008. [Online]. Available: http://www.idav.ucdavis.edu/publications/print_pub?pub_id=936
- [9] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "Gpu cluster for high performance computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, ser. SC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 47–. [Online]. Available: <http://dx.doi.org/10.1109/SC.2004.26>
- [10] V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, and W. mei Hwu, "Gpu clusters for high-performance computing," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, September 2009, pp. 1–8.
- [11] N. Corporation, ["http://www.nvidia.com/object/what_is_cuda_new.html"](http://www.nvidia.com/object/what_is_cuda_new.html), "NVidia Corporation", Tech. Rep., 2011.
- [12] NVIDIA, "Cuda-accelerated applications," NVIDIA Corporation, Tech. Rep., 2005. [Online]. Available: http://www.nvidia.com/object/cuda_app_tesla.html
- [13] T. G. you Group, ["http://gp-you.org/"](http://gp-you.org/).
- [14] A. Corporation, ["http://www.accelereyes.com/"](http://www.accelereyes.com/), AccelerEyes Corporation, Tech. Rep., 2010.
- [15] M. Feldman, "Matlab adds gpgpu support," September 2010. [Online]. Available: <http://www.hpcwire.com/features/MATLAB-Adds-GPGPU-Support-103307084.html>
- [16] M. Inc., "Matlab adds gpgpu support," 2010. [Online]. Available: <http://www.mathworks.com/discovery/matlab-gpu.html>
- [17] J. M. Squyres, "Processes, processors, and MPI, oh my!" *ClusterWorld Magazine, MPI Mechanic Column*, vol. 2, no. 1, January 2004. [Online]. Available: ["http://cw.squyres.com/"](http://cw.squyres.com/)
- [18] *Fast k nearest neighbor search using GPU*, 2008. [Online]. Available: <http://dx.doi.org/10.1109/CVPRW.2008.4563100>

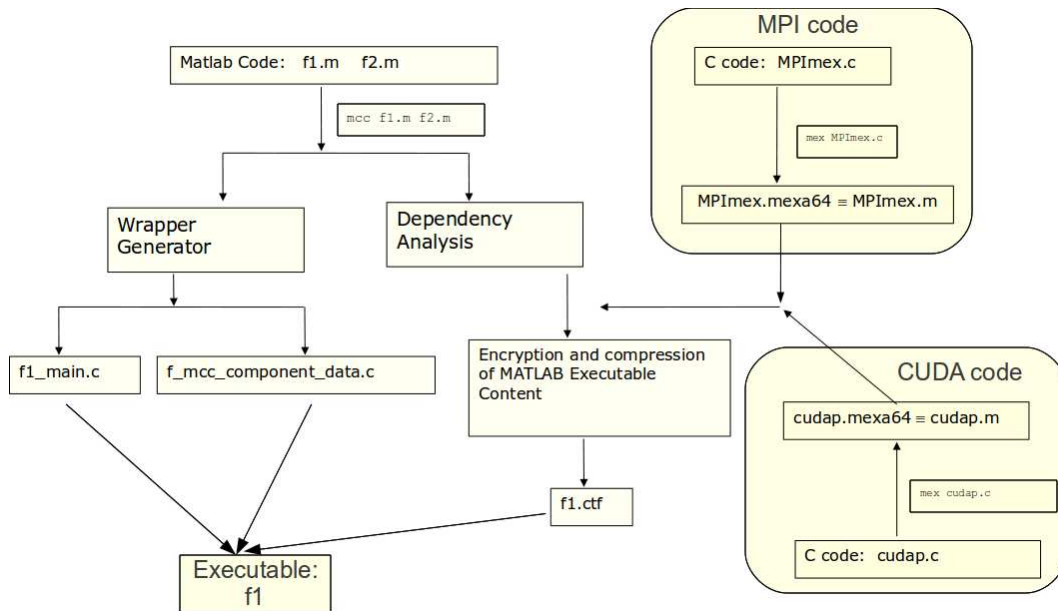


Fig. 3. MATLAB Compiler deployment process for an MPI + CUDA application.

```

%only the root node has two GPUs if(myrank==0) GPUDevice=1;
else GPUDevice=0;
end
...
[ value, indx ] = knn_cuda_with_indexes (ref,query,1, GPUDevice);
...
% Collect solutions
if(myrank==0)
    %alloc space for all the solutions
    soluProc=zeros(1,size_w);
...
    %for each process, if it's not myself, get the closest neighbour
    for l=1:size_w-1,
        soluProc(l)=MPIImex('Recv',1,'MPI_DOUBLE',1,202, 'MPI_COMM_WORLD','IGNORE');
    end
    % Compute the nearest neighbours
    [ value, indx ] = knn_cuda_with_indexes (X(soluProc,:),query,1, GPUDevice);
...
else
    MPIImex('Send',indx, 1, 'MPI_DOUBLE', 0, 202, 'MPI_COMM_WORLD');
end
  
```

Fig. 4. Example of code where the computer which hosts processes 0 and 1 sets the GPUDevice parameter selecting the GPU to be used. Afterwards, it is shown the MPIImex calls to collect the results of the k -NN values computed by the other processes.

A Parallel Domain Decomposition Algorithm for Solving the Equation of Nitric Oxide Diffusion in the Nervous System

Jianxin Wang¹, Heng Wu^{1,2}, and Yu Zhuang^{2,i}

¹College of Information Science & Engineering, South Central University, Changsha, Hunan, China

²Computer Science Department, Texas Tech University, Lubbock, Texas.

Abstract - We present a domain decomposition method for solving the equation modeling nitric oxide diffusions. The domain decomposition we use is one of the stabilized explicit implicit domain decomposition (SEIDD) methods. The SEIDD methods have a restriction that the interface boundaries have no cross-over inside the domain. In this paper, we present a domain-data partition strategy for SEIDD methods and the associated parallel algorithm with parallelism higher than the number of subdomains, overcoming a major accuracy disadvantage of no-crossover interface boundaries for massively parallel processing and enabling high fidelity large scale simulations, as supported by tests with up to 1024 processors.

Keywords: Domain decomposition, parallel algorithms, domain-data partition schemes

1 Introduction

The signaling function of nitric oxide in cardiovascular system is well recognized and its discovery was awarded Nobel Prize in 1998 [11]. It is also discovered that nitric oxide plays a signaling facilitating role in the nervous system, complementing the dominant inter-neuron signaling process through chemical and electrical synapses [12]. The mechanism that nitric oxide functions inter-neuron signaling is diffusion [12,16], which enables communication between neurons that are not connected by synapses or gap junctions (i.e. electrical synapses). The diffusion of nitric oxide in the nervous system is modeled by

$$\frac{\partial u}{\partial t} = \nabla \cdot (D\nabla)u + P(t, x) - S(x)u - \lambda u \quad (1)$$

where x is the vector space variable indicating as spatial location, u is the concentration of the nitric oxide, $D(x)$ is the diffusion coefficient (assuming low concentration of nitric oxide, D is hence assumed to be independent of u but dependent only on the solution environment), P and Q are non-negative functions indicating sources and sinks respectively, and λ is the decay-rate.

In this paper, we present an algorithm for solving equation (1) on parallel computers, which is based on an explicit implicit domain decomposition (EIDD) method in [23]. EIDD [1-5,8-10,13,14,17,18,20-24] are globally non-iterative, non-overlapping domain decomposition methods for solving parabolic equations, which are algorithmically simple, computationally and communicationally efficient for parallel processing. One group of EIDD methods achieves good stability with implicit correction of the explicitly predicted interface boundary conditions. Due to its simplicity, efficiency, and stability, corrected EIDD methods started to receive attention around the turn of the millennium [13,21,3]. One author of this paper studied stabilized EIDD (SEIDD) methods [23], a sub-class of corrected EIDD method in which the predictor and corrector are so designed that the corrector stabilizes the predictor of the next time step.

In parallel implementation of corrected EIDD methods, the correction step is difficult to be parallelized when the interior boundaries cross into each other inside the domain, e.g. as in Figure 1(a). While for some problems [19], it causes no trouble to partition a domain stripwisely with no intersecting interior boundaries, in many cases corrected EIDD methods suffer from low accuracy when partitioned into a large number of narrow strip subdomains when a large number of processors is used [23]. To address the interior boundary crossover problem, Shi and Liao [14] introduced zigzag interior (ZI) boundaries so that in the implicit correction, spatial discretization does not result in coupling of all grid points on the interior boundaries into one single equation. Liao, Shi, and Sun [10] recently developed composite interior (CI) boundaries by replacing the zigzag interior boundaries in [14] with straight-line interior (SI) boundaries at locations not neighboring intersection points of interior boundaries, leading to improved programming simplicity for the treatment of interior boundaries than the ZI boundaries. Zhu, Yuan, and Du [17,18] used a different technique to handle the crossover of interior boundaries. They used special treatment for the implicit discretization at points neighboring intersection points while maintaining unconditional stability. The interface boundary treatment introduced by Jun and Mai for their modified implicit prediction (MIP) method [6,7] can also be used to solve the

intersecting interior boundary problem for corrected EIDD methods.

In this paper, we avoid narrow and long strip subdomains with a different approach in domain partitioning combined with other techniques. The domain partitioning approach was initially studied in [24] with a SEIDD method for the heat equation. The heat equation is separable and the time-discretized equation on each subdomain is solved by a FFT-based parallel solver. This paper considers the nitric oxide diffusion problem, where the equation is not separable and hence the techniques in [24] are not applicable. In this paper we investigate the combination of the domain-partition approach in [24] with a different SEIDD for the nitric oxide diffusion problem.

2 The Stabilized EIDD Method

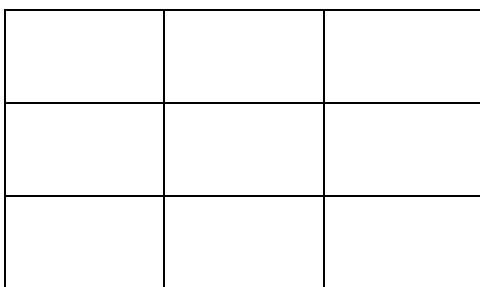
The SEIDD method we use was presented in [23]. For reading convenience, mathematical description of the SEIDD method is provided below. We first list some notations. To numerically solve problem (1), we choose a discrete spatial grid Ω_h with mesh sizes bounded by h , and discretize equation (1) spatially into

$$\begin{cases} \frac{d}{dt}u(t) = A u(t), \\ u(0) = u^0, \end{cases} \quad (2)$$

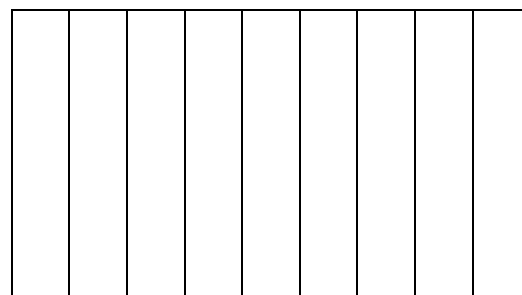
where $A(t)$, a square matrix, is the discrete approximation of the spatial operator on the right hand side of equation (1), and $u(t)$ is the solution vector at time t . Our description will be based on this spatially discrete form of the equation. For a subset S of the discrete grid points Ω_h , i.e. $S \subset \Omega_h$, let χ_S be a diagonal matrix with 1 on the positions corresponding to the grid points in the subset $S \subset \Omega_h$ and 0 elsewhere. For a domain partitioned as in Figure 1 (b), let B be the set of grid points on interface boundaries. With u^k denoting the numerical solution of the k -th time step, the SEIDD method for computing the solution u^{k+1} at the $(k+1)$ -th time step from the current k -th time step is given below.

The SEIDD Method

- 1) Compute the interface boundary condition using the forward Euler scheme



(a)



(b)

Figure 1

$$\begin{cases} \chi_B u^{k+1/3} = \chi_B (I + \Delta t A_h) u^k, \\ \chi_{B^c} u^{k+1/3} = \chi_{B^c} u^k, \end{cases} \quad (3)$$

where I is the identity matrix.

- 2) Using the interface boundary conditions computed at step 1 together with exterior boundary conditions, compute the solution on the subdomains using the directionally factorized implicit scheme

$$\begin{cases} \chi_B u^{k+2/3} = \chi_B u^{k+1/3} \\ \chi_{B^c} (I - \Delta t A_h^x)(I - \Delta t A_h^y) u^{k+2/3} = \chi_{B^c} u^{k+1/3} \end{cases} \quad (4)$$

- 3) Throw away the interface boundary condition computed at step 1, and using solution data $u^{k+2/3}$ on nearby subdomain as boundary conditions, re-compute interface boundary condition on B with the backward Euler.

$$\begin{cases} \chi_B (I - \Delta t A_h) u^{k+1} = \chi_B u^k \\ \chi_{B^c} u^{k+1} = \chi_{B^c} u^{k+2/3} \end{cases} \quad (5)$$

3 The New Parallel SEIDD Algorithm

The method in the previous section is an operator-splitting time discretization method, where the operator splitting is domain decomposition based. On the other hand, data parallelism-based parallel processing involves data partition which, for domain decomposition methods, usually uses the domain partition of the operator- splitting for the data partition. In domain decomposition based parallel algorithms, while the domain partition for constructing the operator-splitting and the domain partition for the parallel- processing-enabling data partition are usually the same, the two partitions, matter-of-factly, do not have to be the same.

For the SEIDD method, when using the same domain partition for the two purposes, the domain partition restriction will prevent parallelism reaching a high level unless very narrow and long strip subdomains are used as in the Figure 1 (b). Very narrow subdomains will cause larger numerical errors as reported in [23]. Thus, to attain high parallelism for the SEIDD methods without using narrow-and-long subdomains, we use the approach of two different domain partitions, which, as applied to the SEIDD method, is detailed as follows

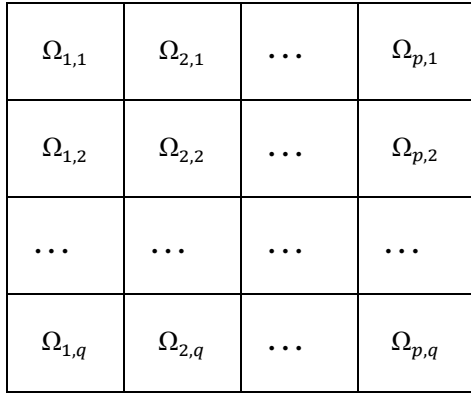


Figure 2

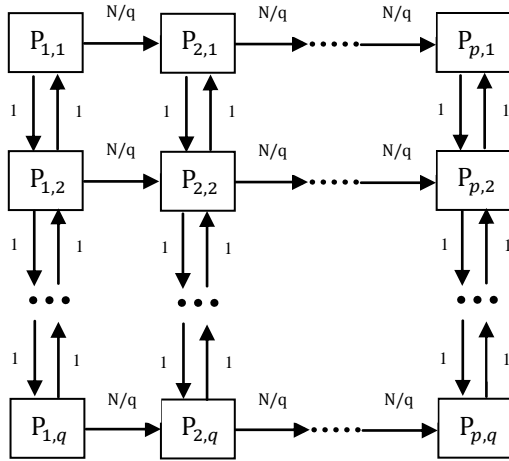


Figure 3: Inter-processor communication pattern and transfer data amount in Step1. Each processor assigned a subdomain of size $M/p \times N/q$.

The domain is divided into $p \times q$ subdomains Ω_{ij} of equal size as in Figure 2. We use the partition by the vertical lines to construct the operator splitting of the SEIDD method while using the partition by both the vertical and horizontal lines for the data partition for distributing data to different processors. To be more specific, let $\Omega_i = \cup_{j=1}^q \Omega_{i,j}$ for $i = 1, \dots, p$, and denote by B_i the vertical interface boundary between Ω_i and Ω_{i+1} for $i = 1, \dots, p-1$. We further let $B = \cup_{i=1}^{p-1} B_i$, and denote by B^c the complement of all vertical interface boundaries in Ω_h . And these two sets B and B^c are the same sets B and B^c used in the construction of the operator-splitting of the SEIDD method presented in Section 2.

We further let B_{ij} denote the part of the interface boundary B_i between subdomains $\Omega_{i,j}$ and $\Omega_{i+1,j}$. Then $B_i = \cup_{j=1}^q B_{i,j}$ for each $i = 1, 2, \dots, p$. Now given $p \times q$ processors labeled as $P_{i,j}$ for $i = 1, 2, \dots, p$ and $j=1, 2, \dots, q$, the data partition and distribution to processors are given as follows:

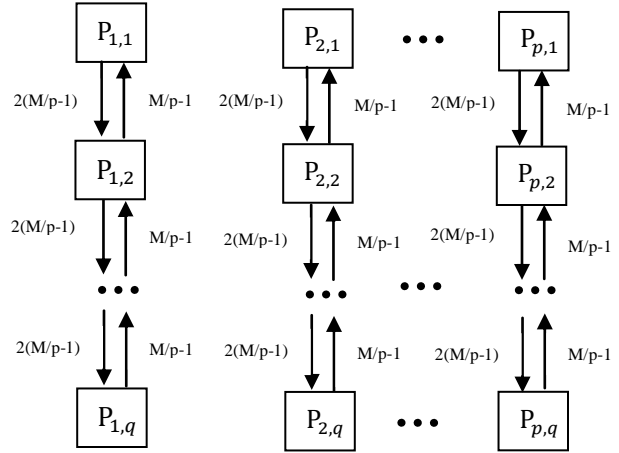


Figure 4: Inter-processor communication pattern and amount of transfer data in Step 2.

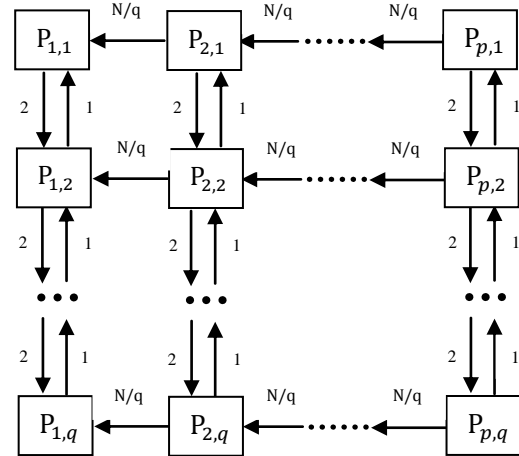


Figure 5: Inter-processor communication pattern and amount of transferred data in Step 3.

- Assign subdomain Ω_{ij} to processor P_{ij} .
- Assign interface boundary B_{ij} to processor P_{ij} .

Step 1 is a matrix-vector multiplication involving the solution only on vertical boundaries, and the matrix is very sparse. Hence Step 1 can be executed by all pq processors in parallel and efficiently.

Step 2 involves solving two sequences of 1-D equations on each of the p vertical subdomains ($i = 1, \dots, p$). One sequence of equations corresponds to $(I - \Delta t A_h^x)$ and the other to $(I - \Delta t A_h^y)$. The two discrete operators $(I - \Delta t A_h^x)$ and $(I - \Delta t A_h^y)$ are diagonally dominant tridiagonal matrices. On the subdomain Ω_i , each tridiagonal system of the sequence that is associated with $(I - \Delta t A_h^x)$ represents a difference equation on a x -direction gridline on Ω_i . Since Ω_i is partitioned along y -direction into q sub-subdomains $\Omega_{i,1}, \Omega_{i,2}, \dots, \Omega_{i,q}$, each x -direction gridline on Ω_i lies entirely on one of the sub-subdomains, say

$\Omega_{i,j}$. Hence each tridiagonal system of the sequence associated with $(I - \Delta t A_h^x)$ has its data entirely on one processor, which allows tridiagonal systems of the sequence $(I - \Delta t A_h^x)$ on different processors be computed completely mutually independently and hence in parallel. Now for tridiagonal systems corresponding to $(I - \Delta t A_h^y)$, due to the y-directional partition of Ω_i into q sub- subdomains, each tridiagonal system of the sequence associated with $(I - \Delta t A_h^y)$ has its data distributed among the q processors $P_{i,1}, P_{i,2}, \dots, P_{i,q}$. Since these tridiagonal matrices are strictly diagonally dominant, they can be solved by the Parallel Diagonal Dominant (PDD) algorithm [15] for tridiagonal matrices.

Step 3 involves solving $p-1$ tridiagonal systems, each corresponding to an equation (5) on one of the $p-1$ interface boundaries B_i for $i = 1, \dots, p-1$. Each of the $p-1$ tridiagonal systems has its data distributed to q processors, e.g. the tridiagonal system corresponding to interface boundary B_i has its data distributed among the q processors $P_{i,1}, P_{i,2}, \dots, P_{i,q}$. Since equation (5) is strictly diagonal dominant, we apply the PDD algorithm for the parallel solution of the tridiagonal systems.

The communications in Steps 1, 2, and 3 of the parallel algorithm are indicated in Figures 3, 4, and 5 respectively, where an arrow indicates the direction of data-transfer between processors and the number on an arrow indicates the number of transferred words in the communication. In Figure 3, the exchange of a pair of words between vertical neighboring processors $P_{i,j}$ and $P_{i,j+1}$ is for the explicit computation of interface boundary condition (in Step 1) at the lower end of $B_{i,j}$ and upper end of $B_{i,j+1}$ respectively. The N/q words sent from processor $P_{i,j}$ to its right neighbor $P_{i+1,j}$ in Figure 3 are the predicted interface boundary condition on $B_{i,j}$, which is needed in Step 2 for solving the subdomain equation on $\Omega_{i,j+1}$ assigned to processor $P_{i+1,j}$. In Figure 4, the data-transfer between two vertical neighboring processors is caused by the PDD tridiagonal solvers, which involves 3 words in communication for each tridiagonal system and a total of $3(M/p-1)$ words between each pair of vertical neighboring processors. In Figure 5, the N/q words sent from processor $P_{i+1,j}$ to its left neighbor $P_{i,j}$ is part of the solution of the subdomain equation on $\Omega_{i+1,j}$ that is needed for the implicit re-computing of u^{k+1} on interface boundary in Step 3; and the data-transfer between two vertical neighboring processors in Figure 5 is caused by the PDD tridiagonal solver of one tridiagonal system in Step 3.

Summarizing the solution process described above, a parallel algorithm is given below:

- (1) For each $i \in \{1, \dots, p-1\}$, processor $P_{i,j}$ send u^k at the lower end of $B_{i,j}$ to its lower neighboring processor $P_{i,j+1}$ for $j < q$, and send u^k at the upper end of $B_{i,j}$ to its upper

neighboring processor $P_{i,j-1}$ for $j > 0$; then processor $P_{i,j}$ explicitly compute the interface boundary condition (IBC) u^{k+1} on $B_{i,j}$ using the forward Euler scheme (3); and then processor $P_{i,j}$ send to its right neighboring processor $P_{i+1,j}$ the just computed IBC.

- (2) Each processor assembles the right hand side of the equation (4) using exterior boundary conditions and the interface boundary conditions computed at step 1. All processors solve tridiagonal systems of associated with $(I - \Delta t A_h^x)$ in parallel; and then for each $i \in \{1, \dots, p\}$, processors $P_{i,1}, P_{i,2}, \dots, P_{i,q}$ combine to solve each of the tridiagonal systems associated with $(I - \Delta t A_h^y)$ on Ω_i using the PDD algorithm.
- (3) For $i = 1, \dots, p-1$, processor $P_{i+1,j}$ send to its left neighboring processor $P_{i,j}$ the solution on the gridline that is on $\Omega_{i+1,j}$ but immediately adjacent to $B_{i,j}$; Then processors $P_{i,1}, P_{i,2}, \dots, P_{i,q}$ combine to re- compute solution u^{k+1} on B_i using the backward Euler scheme (5). The implicit re-computation is carried out by solving the tridiagonal system obtained from the backward Euler scheme using the PDD algorithm.

The upper bounds for computation and communication costs for each of the three steps are given below.

- (1) $12N/q$ parallel floating point operations, and $6\alpha + (N/q + 2)\beta$ communication time, where α is the start-up time for each communication operation and β is the per-word data transfer latency.
- (2) $43(M-p)N/pq$ parallel floating point operation time and $4\alpha + 3(M/p-1)\beta$ communication time, where $19(M-p)N/pq$ of the parallel floating point operations come from the PDD solver for solving y-direction tridiagonal systems, each distributed among q processors, $8(M-p)N/pq$ of the floating point operations come from solving x-direction tridiagonal systems, and $16(M-p)N/pq$ of the floating point operations come from assembling the matrix coefficients for aforementioned the y- and x-direction tridiagonal systems.
- (3) $30N/q$ parallel floating point operation time and $6\alpha + (3 + N/q)\beta$ communication time, and $17N/q$ floating point operations come from solving a y-direction tridiagonal system for each interface boundary B_i distributed among q processors, and $13N/q$ from assembling the matrix and the right hand side the equation.

Summing up all the costs above, the total costs for each time step are approximately $41MN/pq$ parallel floating point operations, and $16\alpha + (3M/p + 2N/q + 2)\beta$ parallel communication time, that is $9MN/pq$ floating point operations more in computation time and $4\alpha + 3\beta M/p$ more in communication time than the existing parallel SEIDD algorithm.

4 Experimental Study

We implemented the parallel SEIDD (PSEIDD) and tested them on an Intel Xeon cluster at Texas Tech High Performance Computing Center with a total of 3360 cores, each at 2.8GHz. The testing problem for the PSEIDD algorithm is the diffusion equation

$$u_t = u_{xx} + u_{yy} + P(t, x, y) - Q(x, y)u$$

with known solution $u(t, x, y) = e^{-2t} \sin(x) \sin(y)$, where $P = [1 - \sin(x) \sin(y)]e^{-2t} \sin(x) \sin(y)$ and $Q = 1 - \sin(x) \sin(y)$. The spatial domain for the testing problem is $[0, 10] \times [0, 10]$, and the simulation time interval is $[0, 1]$. We carried out tests as processor numbers increased from 1 through 1024.

We tested the parallel algorithm on a spatial grid of size 4096×4096 with 4000 simulation time steps. In the tests, the spatial domain is divided into $\sqrt{p} \times \sqrt{p}$ equal-size square sub-subdomains, with p ranging from 1 to 1024, and each square sub-subdomain is assigned to a different processor. We measured the communication time (T-comm, in seconds), the total execution time (in seconds), and the maximal error of the numerical solution at time $t = 1$. These data are listed in Table 1 together with calculated values for Speedup, and Efficiency, where the Speedup and Efficiency are calculated by

$$\text{Speedup}(p) = T_{total}(1)/T_{total}(p),$$

$$\text{Efficiency}(p) = \text{Speedup}(p)/p.$$

The data in Table 1 show two major drops in parallel efficiency when the machine size goes from 1 processor to 1024 processors, one between 1 and 4 processors and the other between 4 and 16 processors. The two drops are due to, in addition to communication overhead, computation overhead of the PDD solver, as explained in Section 3, that the PDD parallel tridiagonal solver in Step 2 of both algorithms incurs $14(M - p)N/pq$ parallel floating point operations for solving y-direction tridiagonal systems when $q=2$, which is $6(M - p)N/pq$ more than the sequential, and incurs $19(M - p)N/pq$ parallel floating point operations for $q \geq 3$, which is $11(M - p)N/pq$ more than the sequential.

Table 1 also show that when machine size goes from 16 to 1024, the efficiency remains almost the same, indicating excellent scalability of the algorithm. This new parallel implementation of the SEIDD algorithm is aimed at large size problems on large size machines (actually for small machine sizes, this new implementation is of no advantage over the existing parallel implementation of SEIDD algorithm as will be shown in the next paragraph), and the testing data suggest high suitability of the new parallelization strategy for solving large problems on large machines.

- Comparison with conventional parallelization

As mentioned in the introduction section that for large machine size, with the conventional parallelization strategy of assigning each subdomain to a different processor will force the entire domain to be partitioned into long and narrow subdomains. Very narrow subdomains would decrease the accuracy of the numerical solutions of SEIDD methods [23]. For comparison, we also solved the same problem with the SEIDD method parallelized in the conventional way with the domain partitioned as in Figure 1 (b). The total execution time and the maximal error are listed in Table 2. The testing data show that the conventionally parallelized SEIDD takes less amount of total execution time than the new algorithm. This is because the PSEIDD algorithm has to use PDD which has a computation cost of $19N$ flops for solving a tridiagonal system of size N while the conventionally parallelized SEIDD algorithm, with each subdomain assigned to one processor, can use the sequential tridiagonal solver which has an $8N$ flops. Though conventional parallelization has advantage in execution time, but their accuracy is much lower when the number of processors reaches 256 and beyond, at which the PSEIDD's is about 14 to 40 times more accurate. And at 64 processors, the accuracies of the two are already very different with the PSEIDD's being about 3 to 4 times more accurate. These testing data suggests that the proposed PSEIDD algorithms are more scalable for massively parallel processing, since with the conventional parallelization, when machine size goes beyond a certain large size, further addition of hardware resources does not contribute to increase of simulation quality.

Table 1: Testing Data of the New Parallel SEIDD Algorithm

Procs	$m \times n$	T-total	T-comm	Speedup	Efficiency	Max-err
1	4096×4096	1.09e+04	3.19e-03	1.0	100%	2.845e-05
4	2048×2048	3.34e+03	1.81e+02	3.3	82.5%	3.909e-05
16	1024×1024	9.47e+02	6.60e+01	11.5	71.9%	3.824e-05
64	512×512	2.38e+02	1.25e+01	45.8	71.6%	3.976e-05
256	256×256	5.97e+01	7.37e+00	183	71.3%	4.800e-05
1024	128×128	1.59e+01	3.80e+00	686	66.9%	6.927e-05

The domain is $[0, 10] \times [0, 10]$ with $h=10/4096$, and the time interval is $[0, 1]$ with $\Delta t = 1/4000$.

The domain divided into $\sqrt{p} \times \sqrt{p}$ subdomains, each with $m \times n$ points, where p is processor number.

The second column under $m \times n$ indicates the subdomain grid size.

Table 2: Comparison of the new parallel SEIDD and the conventionally parallelized SEIDD

New Parallel SEIDD				Conventionally Parallelized SEIDD			
Procs	$m \times n$	T-total	Max-err	Procs	$m \times n$	T-total	Max-err
1	4096×4096	1.09e+04	2.845e-05	1	4096×4096	1.09e+04	2.845e-05
4	2048×2048	3.34e+03	3.909e-05	4	1024×4096	2.93e+03	3.824e-05
16	1024×1024	9.47e+02	3.824e-05	16	256×4096	6.77e+02	4.800e-05
64	512×512	2.38e+02	3.976e-05	64	64×4096	1.66e+02	1.481e-04
256	256×256	5.97e+01	4.800e-05	256	16×4096	4.04e+01	6.685e-04
1024	128×128	1.59e+01	6.927e-05	1024	4×4096	1.24e+01	2.759e-03

The column under $m \times n$ indicate the subdomain grid size, and the number of time steps is 4000.

5 Conclusion

Based on the SEIDD domain decomposition method, we have proposed a new parallelization strategy with which the data partition and distribution to processors do not follow the boundaries of the domain partition that is used to construct the operator splitting of the SEIDD methods. By combining parallelism creation at the temporal level of the SEIDD method with spatial level parallelism utilization, we have achieved higher flexibility and parallelism than allowed by the conventional parallelization technique for domain decomposition methods, while also having avoided violating a requirement of the SEIDD method. The higher data-partition flexibility for parallel processing, as supported by experimental results, improves the quality of large scale simulations on massively parallel systems in capturing better details of the problems under the simulation study.

Acknowledgment

This research used computing resources at Texas Tech High Performance Computing Center.

References

- [1] K. Black. "Polynomial collocation using a domain decomposition solution to parabolic PDE's via the penalty method and explicit-implicit time marching"; J. Sci. Com-put., No. 4, 313-338,7 (1992).
- [2] H. Chen and R. Lazarov. "Domain Splitting algorithm for mixed finite element approximations to parabolic problems"; East-West J. Numer. Math., Vol. 4, No. 2, pp 121-135,1996.
- [3] D. S. Daoud, A. Q. M. Khaliq and B. A. Wade. "A con-overlapping implicit predictor- corrector scheme for parabolic equations"; International Conf. Parallel & Distributed Processing Techniques & Applications (PDPTA 2000), Las Vegas, NV, H.R Arabia et al, ed., Vol. 1, CSREA Press, pp.15-19,2000.
- [4] C. Dawson, Q. Du, and T. Dupont. "a finite difference domain decomposition algorithm for numerical solution of the heat equation"; Math. Comp. 57, No. 195, pp. 63-71,1991.
- [5] M. Dryja. "Sub-structuring methods for parabolic problems"; Forth International Symposium on Domain Decomposition Methods for Partial Differential Equations (Moscow, 1990), pp. 264-271, SIAM, Philadelphia, PA, 1991.
- [6] Y. Jun and T.-Z. Mai. "ADI method – domain decomposition"; Applied Numerical Math., Vol. 56, pp. 1092-1107,2006
- [7] Y. Jun and T.-Z. Mai. "Numerical analysis of the rectangular domain decomposition method"; Communications in Numerical Methods in Engineering, Vol. 25, pp. 810–826, July 2009.
- [8] Y. A. Kuznetsov. "New algorithms for approximate realization of implicit difference schemes"; Sov. J. Numner. Ana. Math. Modell., pp 99-114, 3 (1988).
- [9] Y. M. Laevsky. "Explicit implicit domain decomposition method for solving parabolic equations"; Computing methods and technology for solving problems in mathematical physics (Russian), pp. 30-46, Ross. Akad. Nauk Sibirsk. Otdel., Vychisl. Tsent, Novosibirsk, 1993.
- [10] H. L. Liao, H. S. Shi, and Z. Z. Sun. "Corrected explicit-implicit domain decomposition algorithms for two-dimensional semilinear parabolic equations"; Science in China, Series A: Mathematics, Vol. 52, No. 11, pp. 2362-2388,2009.
- [11] The Nobel Foundation, The Nobel Prize in Physiology or Medicine 1998. Nobelprize.org, http://nobelprize.org/nobel_prizes/medicine/laureates/1998/illpres/.
- [12] A. Philippides, P. Husbands, T. Smith, and M. O'Shea. "Structure-based models of NO diffusion in the nervous system, Computational Neuroscience: a Comprehensive approach (J. Feng ed.)"; Chapman & Hall/CRC, London, pp. 97-130,2004.
- [13] H. Qian and J. Zhu. "On an efficient parallel algorithm for solving time dependent partial differential equations"; Proceeding of the 11th International Conference on Parallel and Distributed Processing Techniques and

- Applications, Las Vegas, CSREA Press, Athens, GA, pp. 394-401, July 1998.
- [14] H.S. Shi and H.L. Liao. "Unconditional stability of corrected explicit-implicit domain decomposition algorithms for parallel approximation of heat equations"; *SIAM. J. Numer. Anal.*, Vol. 44, pp.1584–1611,2006.
- [15] X.-H. Sun, H. Zhang, and L. Ni. "Efficient Tri-diagonal Solvers on Multi-computer"; *IEEE Trans on Computers*, Vol. 41, pp 286-296, 1992.
- [16] J. Wood and J. Garthwaite. "Models of the diffusional spread of nitric oxide: Implications for neural nitric oxide signaling and its pharmacological properties"; *Neuropharmacology*, Vol. 33, No. 11, pp. 1235-1244, November 1994.
- [17] L. Zhu, G. Yuan, and Q. Du. "An explicit-implicit predictor-corrector domain decomposition method for time dependent multi-dimensional convection diffusion equations"; *Numer. Math. Theor. Meth. Appl.*, Vol. 2, pp. 1-25,2009.
- [18] L. Zhu, G. Yuan, and Q. Du. "An efficient explicit/implicit domain decomposition method for convection-diffusion equations"; *Numerical Methods for Partial Differential Equations*, Vol. 26, No. 4, pp. 852–873, July 2010.
- [19] Y. Zhuang. "A parallel and efficient algorithm for multi-compartment neuronal modeling"; *Neurocomputing*, vol. 69, Issues 10-12, pp. 1035-1038, June 2006.
- [20] Y. Zhuang. "An alternating explicit implicit domain decomposition method for the parallel solution of parabolic equations"; *Journal of Computational and Applied Mathematics*, Vol. 206, no. 1, pp. 549-566,1991.
- [21] Y. Zhuang and X.-H. Sun. "A domain decomposition based parallel solver for time dependent differential equations"; In: *Proc. 9th SIAM Conf. Parallel Processing for Scientific Computing*, March 1999, San Antonio, Texas. CD-ROM SIAM, Philadelphia, 1999.
- [22] Y. Zhuang and X.-H. Sun. "Stable, globally non-iterative, non-overlapping domain decomposition parallel solvers for parabolic problems"; *Proceeding of ACM/IEEE Super Computing Conference*, Denver, Colorado. CD-ROM IEEE Computer Society and ACM, November 2001.
- [23] Y. Zhuang and X.-H. Sun. "Stabilized explicit implicit domain decomposition methods for the numerical solution of parabolic equations"; *SIAM J Sci. Comput.*, Vol. 24, No 1, pp. 335-358, July 2002.
- [24] Y. Zhuang and X.-H. Sun. "A highly parallel algorithm for the numerical simulation of unsteady diffusion processes"; *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, Denver, Colorado, April 2005.

Accelerating the Computation and Verification of Molecular Collision Models: A Case Study in Legacy Code Parallelization

Kurt A. O'Hearn[†], Christian Trefftz[†], George C. McBane[‡], and Gregory Wolffe[†]

[†]School of Computing and Information Systems, Grand Valley State University, Allendale, MI, USA

[‡]Department of Chemistry, Grand Valley State University, Allendale, MI, USA

Abstract—*This research project constituted a case study in computational science: applying modern parallel computing techniques to a mathematical model used to solve a scientific problem. The problem involved a physical chemistry model that evaluates simulations of molecular collision experiments, implemented via a 15,000-line FORTRAN 77 code. This problem was chosen for parallelization because of its scientific importance, its computational complexity, and its overall structure that was amenable to parallelization. Since the original program was written, experimental designs have changed in a way that require significant increases in execution time for the simulation. To address this issue, the simulation code was profiled, analyzed, and parallelized using the OpenMP/multithreaded paradigm. Nearly linear speedup was measured for the OpenMP version executing on a 16-core multiprocessor. Furthermore, experimental results suggest these speedups should scale well with an increasing number of processors.*

Keywords: Computational science, OpenMP, FORTRAN, molecular dynamics, parallelization

1. Introduction

Collision experiments provide valuable information about the forces between molecules, that is, about the potential energy surfaces that describe their interactions. Many fields of science depend on accurate data about molecular interactions. For example, in astrophysics, a better understanding of the collisions of the most common atoms and molecules in space, containing carbon, nitrogen, helium, hydrogen, and oxygen, has resulted in more accurate interpretations of the molecular rotational spectra obtained from space telescope observations [6].

The potential energy surface for a given pair of molecules is not directly observable. The primary observable quantity in several types of collision experiment is the *differential cross section* (DCS), $\frac{d\sigma}{d\omega}(\theta)$, which describes the probability distribution of the scattering angle θ for collisions producing a particular set of products. The scattering angle is the angle between the entering and the exiting velocity vectors of one collider in the center-of-mass frame. The DCS can be computed from an assumed or computed potential energy surface, and can also be extracted from data obtained in

collision experiments. It therefore provides a primary point of comparison between experiment and theory.

The scientific problem studied in this project was the modeling and simulation of molecular collision experiments for the purpose of extracting the DCS from experimental data. The extraction takes the form of a fitting procedure: assume a DCS, apply a simulation that models the characteristics of the experimental apparatus to predict the resulting data set, and adjust the DCS until the predicted data match the experimental results. The models were originally implemented as a 15,000-line FORTRAN 77 simulation code. The goals in rewriting and parallelizing this code were as follows:

- *Accuracy:* The original code employed approximate interpolation methods to reduce the number of computations performed. The parallelized code eliminates these interpolations, thereby improving the accuracy of the simulation.
- *Features:* Use of the interpolation method also prevented the simulation from being extended to describe a new experimental configuration. Eliminating the interpolation will permit these extensions to be implemented.
- *Speed:* However, removing the interpolations incurs a significant increase in computational cost, as many more models must be evaluated. Additionally, the modifications required to accommodate the new experimental design also increase computation. Without parallelization, the simulation would take many hours to run. Our primary goal in parallelizing the code was to make the simulation radically faster.

By developing a parallelized simulation that is more accurate and an order of magnitude faster than the original, we hoped to broaden the applicability of this code.

In addition to describing our method and results, this paper presents our experiences as a case study in parallelizing a legacy scientific code. Section 2 describes the physical chemistry experiment modeled by the FORTRAN simulation. Section 3 describes the development platform and the problem decomposition. Section 4 documents the parallelization process and the challenges encountered. Section 5 presents and discusses our results along with tips for legacy code parallelization. The final section summarizes conclusions drawn and discusses directions for future research.

2. Problem Domain: Physical Chemistry

The collision experiments are performed by intersecting two pulsed molecular beams and detecting evidence of resulting collisions using laser ionization of the collision products [5]. The simulation models a crossed molecular beam machine with resonance enhanced multiphoton ionization (REMPI) for spectroscopic detection and subsequent velocity mapping, as depicted in Fig. 1 [1].

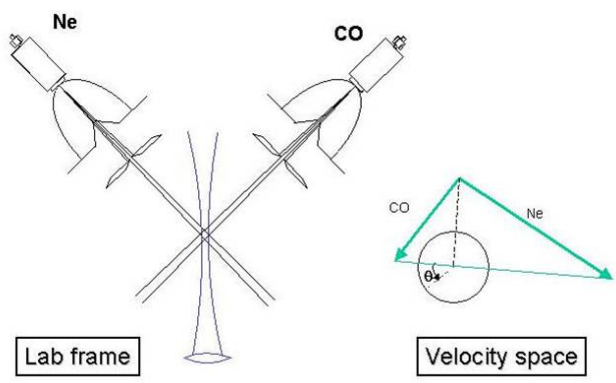


Fig. 1: Left side: spatial arrangement of the experiment. The two valves producing the molecular beams (of CO and neon in this example) are open for a time on the order of $100 \mu s$ before the few-nanosecond laser pulse. The resulting CO ions are accelerated upward out of the plane of the figure toward an imaging detector whose active surface is parallel to the figure plane. Each ion strikes the detector at a point determined by its postcollision lab frame velocity and independent of its position at ionization. Right side: Newton diagram modeling the velocity vectors of the colliders. The intersection point at the top of the triangle represents zero laboratory velocity. For given speeds in the two beams, all the product molecules in a particular internal state will have final velocity vectors on the surface of the “Newton sphere” shown as a circle in the figure. The differential cross section describes the intensity distribution on that sphere as a function of the scattering angle θ . The distribution is cylindrically symmetric around the relative velocity vector (the hypotenuse of the triangle) in the center of mass frame.

The result of this type of experiment, and of the simulation, is an image as seen in Fig. 2. The intensity in each pixel represents the likelihood of an ion striking the detector at the corresponding location. If the ionization probability was independent of the laboratory position or velocity of the product molecule, the differential cross section could be read directly from the image as the intensity as a function of angular position around the circle. Instead, the laser ionization detection is more sensitive to molecules that move slowly in the laboratory, or along the laser direction. For that reason the image appears asymmetric, and a careful simulation is needed.

To extract the differential cross section from the image, an assumed set of parameters describing the DCS and a few hard-to-measure characteristics of the experiment are used to simulate an image. The actual image from the experiment is then compared with the image generated by the simulation, and the parameters adjusted iteratively to obtain a good match.

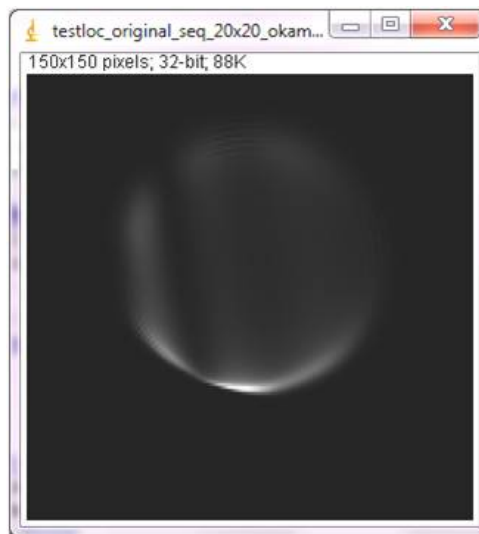


Fig. 2: Image generated by one simulation, as displayed by the ImageJ program [7]. The Newton diagram is inverted with respect to Fig. 1: the pixel corresponding to zero lab-frame velocity is near the bottom center of the image, and the relative velocity vector slopes slightly upward from left to right.

In the simulation, the intensity of a particular pixel whose center corresponds to lab frame velocity components \bar{v}_x and \bar{v}_y is evaluated according to [5]

$$I(\bar{v}_x, \bar{v}_y) \propto \iint dv_A dv_B g_A(\mathbf{v}_A) g_B(\mathbf{v}_B) \times g \left(\iint_{\text{pixel}} dv_x dv_y J(\mathbf{v}_+) \right) \int_{-\infty}^0 dt \int_{V_{\text{coll}}} d\mathbf{r} \times \left[P_1(\mathbf{r}, \bar{\mathbf{v}}_+) n_A(\mathbf{r} + \bar{\mathbf{v}}_+ t, t) \times n_B(\mathbf{r} + \bar{\mathbf{v}}_+ t, t) \frac{d\sigma}{d\omega}(\bar{\mathbf{v}}_+) + P_1(\mathbf{r}, \bar{\mathbf{v}}_-) \times n_A(\mathbf{r} + \bar{\mathbf{v}}_- t, t) n_B(\mathbf{r} + \bar{\mathbf{v}}_- t, t) \frac{d\sigma}{d\omega}(\bar{\mathbf{v}}_-) \right]. \quad (1)$$

The central idea behind this probability calculation is that the number of ions whose final velocity vector lies in a small range defined by one pixel can be obtained by integrating backward in time, considering all the collisions that could have produced a product molecule in the ionization volume

at the time the laser fired. v_A and v_B are the precollision velocities of the colliding molecules, $g_A(v_A)$ and $g_B(v_B)$ are their distributions in the molecular beams, and v_x and v_y are the components of the postcollision laboratory frame velocity of the scattered A molecule in the plane of the detector. The initial relative speed is $g = |v_A - v_B|$, and t is the difference between the time of the scattering event and the firing of the probe laser. The values \bar{v}_+ and \bar{v}_- represent the two possible laboratory frame velocities, on the upper and lower halves of the Newton sphere, for a scattered molecule whose v_x and v_y correspond to the center of the pixel. They depend implicitly on v_A and v_B because those determine the position of the Newton sphere. $P_1(\mathbf{r}, \mathbf{v})$ is the ionization probability for an A molecule at position \mathbf{r} with velocity \mathbf{v} when the probe laser fires. This probability depends on \mathbf{r} through the spatial dependence of the laser intensity and on \mathbf{v} through the Doppler shift. The functions $n_A(\mathbf{r}, t)$ and $n_B(\mathbf{r}, t)$ represent the densities of A and B molecules in the molecular beams at position \mathbf{r} and time t . They appear with shifted position arguments in the intensity equation, as $n_A(\mathbf{r} + \mathbf{v}t, t)$, because molecules scattered at time t (which is negative) and position $\mathbf{r} + \mathbf{v}t$ will arrive at position \mathbf{r} to be ionized at time $t = 0$. $J(\mathbf{v})$ gives the Jacobian for the projection of the Newton sphere onto the planar detector; it is proportional to $(|\mathbf{v} \cdot \mathbf{e}_z|)^{-1}$, where \mathbf{e}_z is the unit vector perpendicular to the detector face. Finally, $\frac{d\sigma}{d\omega}(\bar{\mathbf{v}})$ is the differential cross section [4].

The only term in the equation that rapidly varies within a single pixel is the Jacobian $J(\bar{\mathbf{v}}_+)$, and the integral of J over the small range of v_x and v_y covering a single pixel can be done analytically. The factor enclosed in large parentheses can therefore be treated as a known function of v_A and v_B for each pixel. The functions g_A , g_B , P_1 , n_A , and n_B are all known from the properties of the apparatus, and the differential cross section $\frac{d\sigma}{d\omega}$ takes an assumed form at the start of each simulation. The basic computational task is therefore the numerical evaluation of a ten-dimensional integral: three dimensions each from the initial velocities v_A and v_B and the ionization position \mathbf{r} , and one from the integration over time. In the original experiment for which the program was developed, the two molecular beams had very small angular divergences, so that the integrations over v_A and v_B could be reduced to a single dimension each. The resulting six-dimensional integration was performed by nested quadratures, using appropriate Gaussian quadratures for the inner four dimensions and the trapezoidal rule for the outer two. In more recent experiments, the molecular beams have larger divergences, and the full ten-dimensional integration is needed. This need for additional nested quadratures motivated our work.

The molecular collision simulation contains various experimental and molecular parameters. Notable parameters include:

- *detxsize* and *detysize*: the numbers of pixels of the ion-

ization detector in the x - and y -directions, respectively. Their product is the number of pixels in the simulated image.

- *avpts* and *bvpts*: the number of quadrature points in velocity for molecules A and B, respectively. Their product determines the number of different relative velocity vectors that must be considered, and hence the number of times that the inner four-dimensional integral over \mathbf{r} and t must be evaluated for each pixel.

3. Platforms and Parallel Design

The paradigm chosen to parallelize the original simulation code was OpenMP/multithreading running on multicore processors [2]. The rationale for using OpenMP was to achieve the benefits of parallelism through multithreading without any requirement of specialized hardware, instead exploiting the multicore chips already found in modern systems.

3.1 Development and Testing Platforms

The OpenMP version was developed and tested on various multicore architectures, ranging from dual-core PCs to a 16-core multiprocessor. Various flavors of Linux were used (Fedora Core and Red Hat Enterprise), as were open source and commercial compilers (the GNU FORTRAN compiler `gfortran` and the PGI® FORTRAN compiler from The Portland Group, Inc.). Code correctness and compatibility were maintained across all development environments.

3.2 Design and Problem Decomposition

A multithreaded program design was chosen due to the independent nature of the pixel intensity calculations. The calculations for each pixel share no data dependencies, hence individual threads can be used to safely generate groups of unique pixels. Furthermore, the pixel intensity calculations appear near the top level of a deep call graph; this allowed specification of the OpenMP directives without fear of recursive routines invoking the parallel region. The problem decomposition among the pixel intensities resulted in a coarse granularity: each thread was responsible for calculating one pixel of the 150 by 150 pixel output image, with the resulting computational complexity of each pixel being relatively high.

4. Case Study: Parallelization Process

The parallelization process used in this project consisted of four distinct phases:

- Satisfying the necessary prerequisites before beginning parallelization
- Profiling the serial code to elucidate effective parallel design options
- Implementing the parallel design options and resolving issues inhibiting implementation
- Developing a testing procedure to verify parallel code correctness

4.1 Prerequisites to Parallelization

Any computational science project begins with the essential prerequisite of acquiring a sufficient background in the scientific domain, necessary to understand the problem being solved. In this case that meant becoming familiar with the principles of scattering experiments with spectroscopic detection, accomplished via a comprehensive review of the literature along with regular consultation with our domain expert.

Often, working with legacy code requires a second prerequisite: understanding the intricacies of the specific programming language used, especially pertaining to data structures and memory management. If the developer is new to the language, a high-quality language reference is an invaluable tool. For this project, a good portion of time was devoted to learning FORTRAN 77 using the science/engineering-focused book by Kuperschmid [3].

A third prerequisite to parallelization is a thorough analysis of the original serial program. In computational science, it is not uncommon to find code written by a domain expert, someone perhaps not versed in the practices of software engineering. In this project, several problems were uncovered that affected code maintainability and version management.

4.2 Code Profiling and Parallel Design

The next phase of the project was to profile the serial simulation, with specific focus on those portions of code that required the longest execution time, and their accompanying data structures. Profiling utilities included:

- The GNU profiler, `gprof`, for generating routine call graph information and measuring the execution time accounted for by each routine
- The GNU debugger, `gdb`, for runtime analysis of the code

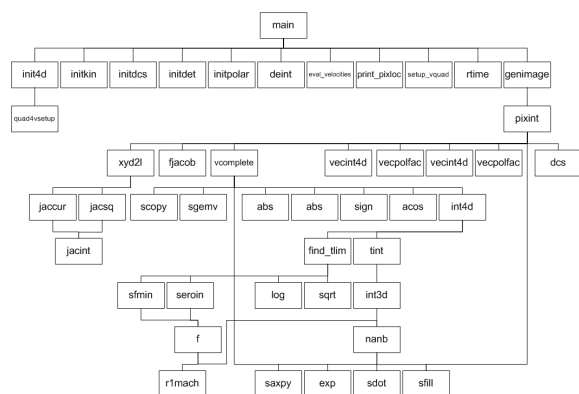


Fig. 3: Partial routine call graph of the code

Profiling information was used to construct and analyze the call graph in Fig. 3. The profiling results indicated a large amount of time was spent executing routines from the Basic

Linear Algebra Subprograms (BLAS) library, which suggested parallelizing the code at a low level in the call graph. However, these routines were invoked at multiple levels and instances in the call graph; parallelization would involve overhead as multiple threads would need to be continuously spawned and destroyed. Furthermore, these routines were already highly optimized to the machine architecture. Any small speedup obtained through parallelization would be likely lost to the thread management overhead.

Alternatives to BLAS parallelization were available because the majority of the BLAS calls were within a code branch performing the pixel intensity calculations. The two candidates within this branch were the pixel intensities region, which was at a high level in the call graph, and the quadrature points region, which was a few levels below the pixel intensities. Neither option contained recursive or divergent paths; thus, OpenMP parallel loop pragmas could be inserted around either section without issue.

Considerations in choosing one design over the other included memory usage, overhead, and the amount of computation available to parallelization. The number of pixel intensity calculations in our sample data (22500) outnumbered the number of quadrature point calculations (as determined by *avpts* and *bvpts*). Because they are situated at a higher level in the call graph, parallelization of the pixel intensity computations would require more memory. However, they would not incur as much thread management overhead as the repeated invocations of the quadrature points code region. Since the simulation did not use a large amount of data, the pixel intensity region was chosen for parallelization. This problem decomposition resulted in a coarse granularity, where each pixel of the 150 by 150 pixel image corresponded to an independent parallel computation.

4.3 Parallel Implementation

The data structures within the pixel intensity branch were analyzed for data dependencies. Two potential issues were found: static variables and COMMON blocks. A single static variable was discovered as part of a root-finding routine. The variable stored the most recent result as a starting point for the next search; this approach could cause inconsistent results if the routine was re-entered by multiple threads working on different parts of the image. This variable was removed, and the root search performed from the beginning for each call, at some cost in computational efficiency.

The other problem stemmed from the use of COMMON blocks. Ten COMMON blocks were contained in ten separate files. Each COMMON block was given scope into particular routines by use of an include statement. Most variables in the COMMON blocks were calculated once during initialization routines prior to the pixel intensity calculations, and subsequently accessed only for reads. However, one array holding the vector velocities of the molecules in the molecular collision experiments was written to during the

pixel intensity calculation, and thus could cause concurrency issues if multiple threads performed concurrent writes. The solution was to remove the velocity array from the COMMON block, locally declare the vector in the `genimage` routine above the pixel intensity calculation branch in the call graph, and pass the array down the call graph as an argument for all routines that required it. This approach ensured that each pixel computation would have a local velocity array, hence be safe for concurrent execution.

4.4 Testing

A systematic testing approach was employed, where all the experimental parameters were held constant between runs while only the number of executing threads varied. Notable values set for parameters include the number of velocity quadrature points for each beam (20) and the number of pixels for each direction in the image (150).

Even after ensuring identical parameters, verifying the correctness of the image output between the serial and parallel code versions was problematic because of changes in the computations. These differences stemmed from two sources: the reliance on interpolation methods and static variables. The original sequential code relied on interpolation methods to estimate the inner four-dimensional integral for many relative velocities, while the parallel version computed the integral directly for every relative velocity vector. Thus, no direct numeric comparison between the interpolated serial and non-interpolated parallel versions could be employed. However, visual and statistical comparisons between the output image files were performed.

Additionally, the removal of static variables fundamentally changed the computations in the code. To verify this change did not invalidate the results, a simple program was written to calculate the maximum difference between corresponding pixels in two image output files. After multiple runs, an average difference of less than 0.5% was observed and regarded as acceptable.

5. Results and Discussion

After verifying the correctness of the results, timing experiments were conducted on a 16-core multiprocessor.

5.1 Performance Results

Table 1 shows the data from the test runs.

Number of Processors	Execution Time (s)	Speedup
1	1454.66	-
2	766.87	1.89
4	388.03	3.74
8	199.83	7.27
16	105.23	13.82

Table 1: Performance data on a 16-core multiprocessor

The execution time column is the total measured execution time for a run using the low resolution wallclock `time`

utility in linux/unix systems. The run with one processor was obtained using the original sequential version of the simulation without interpolation, and all other runs used the parallel version. The speedup calculation was computed as the ratio of the sequential run time against particular parallel run times.

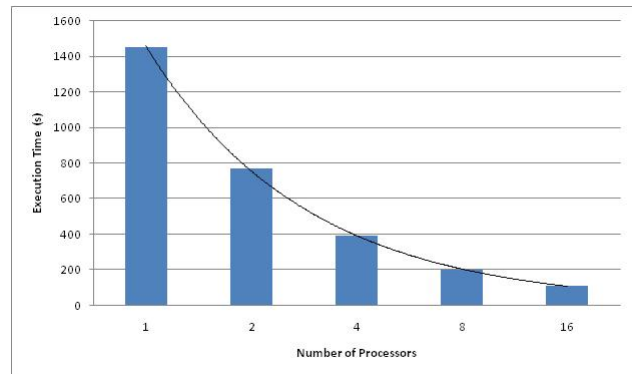


Fig. 4: Performance metric: total execution time

Fig. 4 indicates that the execution time of the simulation was reduced from approximately twenty-five minutes to less than two minutes executing on sixteen cores; this amounts to a nearly fourteen times speedup in total execution time. The regression line illustrates the decrease in execution time with the increase in the number of processors. The primary result of performance testing: nearly linear speedup was measured for the parallel version of the code.

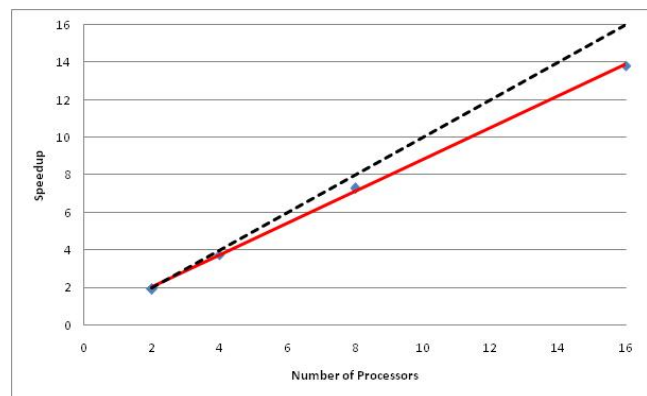


Fig. 5: Performance metric: speedup

In Fig. 5, the black dotted line indicates the theoretically optimum speedup (i.e., using n processors results in an n -times speedup). Our parallel solution, indicated by the solid red line, approaches this optimum. The linear nature of the parallel solution suggests speedups should continue to scale well with an increasing number of processors.

5.2 Tips and Insights: Legacy Parallelization

Legacy code offers many unique challenges during the parallelization process. The following list presents some

reminders and insights for success.

- 1) *Parallelization obstacles*: Begin profiling by identifying the following potential inhibitors to parallelism: global and static variables. Also, examine the code for deprecated language constructs that may cause unexpected behavior parallelized.
- 2) *Platforms and support*: Be sure to upgrade to the latest version of compilers and libraries if support for legacy code is available. Also, be aware of libraries that are not known to be thread-safe.
- 3) *Utilities*: Quality profiling tools can be an invaluable tool in deciphering convoluted legacy code. Tools that may be employed include thread checkers, memory tracers, and visual profilers.

6. Conclusions and Future Work

An OpenMP multithreaded version of a molecular collision simulation code was created and statistically verified for correctness. The new, parallel code drastically reduced execution time compared to the original program, demonstrating nearly linear speedup and good scalability.

Additionally, the parallel version improved the accuracy of the simulation by eliminating an interpolation approximation.

As a result of replacing interpolation with direct computation, the program can now be modified to work with new experiments. We plan to further modify the program to make use of accelerator chips such as FPGAs and GPUs, and possibly hybrid approaches.

References

- [1] A. T. J. B. Eppink and D. H. Parker, "Velocity map imaging of ions and electrons using electrostatic lenses; application in photoelectron and photofragment ion imaging of molecular oxygen," *Rev. Sci. Inst.*, vol. 68, pp. 3477-3484, 1997.
- [2] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald, *Parallel Programming in OpenMP*, Morgan Kaufman Publishers, 2001.
- [3] Michael Kupferschmid, *Classical Fortran: Programming for Engineering and Scientific Applications*, 2nd ed., Marcel Dekker, Inc. January 14, 2009.
- [4] K. T. Lorenz, D. W. Chandler, G. C. McBane, "State-to-State Differential Cross Sections by Velocity Mapping for Rotational Excitation of CO by Ne," *J. Phys. Chem*, 2002, 106 (7), 1144-1151.
- [5] K. T. Lorenz, M. S. Westley, D. W. Chandler, "Rotational State-to-State Differential Cross Sections for the HCl-Ar Collision System Using Velocity-Mapped Ion Imaging," *J. Phys. Chem*, 2000, 2, 481-494.
- [6] C.-H. Yang, G. Sarma, J. J. ter Meulen, D. H. Parker, G. C. McBane, L. Wiesenfeld, A. Faure, Y. Scribano, and N. Feautrier, "Communication: Mapping Water Collisions for Interstellar Space Conditions", *J. Chem. Phys.*, 133, 131103 (2010), DOI:10.1063/1.3475517.
- [7] W. S. Rasband, ImageJ, U. S. National Institutes of Health, Bethesda, Maryland, 1997-2011, <http://imagej.nih.gov/ij/>.

A Safety-strengthened Election Protocol Based on an Unreliable Failure Detector in Distributed Systems

Yong-Hwan Cho, Seon-Hyong Lee, Yeong-Mok Kim and Sung-Hoon Park
Dept. of Computer Science, Chungbuk National University, Chung-Buk 330-800, Korea

Abstract

A Leader is a Coordinator that supports a set of processes to cooperate a given task. This concept is used in several domains such as distributed systems, parallelism and cooperative support for cooperative work. In completely asynchronous systems, there is no solution for the election problem satisfying both of safety and liveness properties in asynchronous distributed systems. Therefore, to solve the election problem in those systems, one property should be weaker than the other property. If an election algorithm strengthens the safety property in sacrifice of liveness property, it would not nearly progress. But on the contrary, an election algorithm strengthening the liveness property in sacrifice of the safety property would have the high probability of violating the safety property. In this paper, we presents a safety strengthened Leader Election protocol with an unreliable failure detector and analyses it in terms of safety and liveness properties in asynchronous distributed systems.

Keywords : Distributed Computing, Leader Election, Asynchronous Distributed Systems, Failure Detectors

1. Introduction

Distributed systems consist of groups of processes that cooperate in order to complete specific tasks. A *Leader* is a Coordinator that supports a set of processes to cooperate a given task. This concept is used in several domains such as distributed systems, parallelism and cooperative support for cooperative work.

To elect a *Leader* (or Coordinator) in a distributed system, an *agreement* problem must be solved among a set of participating processes. This

problem, called the *Election* problem, requires the participants to agree on only one leader in the system [1]. The problem has been widely studied in the research community [2,3,4,5,6]. One reason for this wide interest is that many distributed protocols need an election protocol.

The Election problem is described as follows. At any time, there is at most one process that considers itself a *leader* and all other processes consider it as to be their only leader. If there is no leader, a leader is eventually elected.

The so-called FLP impossibility result, which states that it is impossible to solve any non-trivial agreement in an asynchronous system even with a single crash failure, also applies to the election problem [7]. That means that there is no solution for the election problem satisfying both of safety and liveness properties in completely asynchronous distributed systems.

It must be pointed out, however, that the impossibility result really means “not always possible,” as opposed to “never possible.” As a matter of fact, any algorithm that tries to solve the Election Problem cannot always make progress without violating safety; there exist cases in which the algorithm violating safety, although it is very unlikely.

Therefore, to solve the election problem in those systems, one property should be weaker than the other property. If an election algorithm strengthens the safety property in sacrifice of liveness property, it would be difficult to progress. But on the contrary, an election algorithm strengthening the liveness property in sacrifice of the safety property would have the high probability of violating the safety property. There exists a trade-off between safety property and

liveness property.

A stable election protocol, which implies the safety strengthened election protocol, is needed in a practical distributed computing environment. Consider a mission critical distributed system such as an electronic commerce system that runs multiple servers in which one of them roles a master (leader) and others are slaves.

To have data consistency among the servers in the system, this system should not violate safety property, which means that all processes connected the system never disagree on a *leader*. In those systems the safety property is more important property than the liveness property.

As a classic paper, there is Garcia-Molina's Invitation algorithm to solve election problem in asynchronous distributed systems. The algorithm strengthens the progress property rather than safety and it allows more than two leaders in the systems.

Our idea is based upon the Garcia-Molina's Invitation algorithm for solving the election problem in asynchronous distributed systems [2]. He redesigns the Bully algorithm for synchronous distributed systems into the Invitation algorithm for asynchronous distributed systems by using a specification that is weak enough to be solvable, allowing the algorithm to progress even in completely asynchronous distributed systems.

His specification uses a strong progress requirement, allowing executions in which even a single process suspicion of the current leader's crash and its attempted leader election from the members may lead a progress to elect a new leader from all processes.

We propose an election algorithm that requires processes to elect a new leader only when they agree with the current leader's crash. This requirement is strong because, if no set of processes agrees on the current leader's crash, no progress is made. The requirement is, however, much more stronger than the one proposed by Garcia-Molina's Invitation algorithm in that it implicitly states that the leader election of any process be allowed only on the basis of only it's own knowledge.

In this paper, we presents a safety strengthened

Leader Election protocol with an unreliable failure detector and analyses it in terms of safety and liveness properties in asynchronous distributed systems.

Our algorithm, based on a standard three phases commit protocol, is fully distributed. It does not extend the asynchronous model of concurrent computation to include global failure detectors. Progress of the algorithm can be guaranteed only in case of minimal violating a safety property.

The rest of the paper is organized as follows. In Section 2, we describe our system model and definitions. In Section 3, this paper relates the election specification to other ways to solve the election problem. In Section 4, this paper provides a stable algorithm that solves the Leader Election problem. In Section 5, we ensure the correctness of the algorithm by proving that it satisfies the two properties of the specification given in Section 4. Finally, Section 6 summarizes the main contributions of this paper and discusses related and future works.

2. Model and Definitions

Our model of asynchronous computation with failure detection is the one described in [9,10]. In the following, we only recall some informal definitions and results that are needed in this paper.

2.1 Processes

We consider a distributed system composed of a finite set of processes $\Omega = \{p_1, p_2, \dots, p_n\}$ where processes are identified by unique id's. Communication is by message passing, *asynchronous* and *reliable*. Processes fail by crashing; Byzantine failures are not considered.

Every pair of processes is connected by a communication channel. That is, every process can send messages to and can receive messages from any other. We assume processes are able to probe a communication channel for incoming messages. Communication channels are considered to be reliable, FIFO, and to have an infinite buffer capacity. A reliable channel ensures

that a message, sent by a process p_i to a process p_j , is eventually received by p_j if p_i and p_j are correct (i.e. do not crash).

Asynchrony means that there is no bound on communication delays or process relative speeds. A process that has been infinitely slow for some time and has been unresponsive to other processes may become responsive again at any time. Therefore, processes can only suspect other processes to have crashed, using local failure detectors.

A failure detector is a distributed oracle which gives hints on failed processes. We consider algorithms that use failure detectors. Local failure detectors are assumed to be inaccurate and incomplete. That is, local failure detectors may erroneously suspect that other, operational processes have crashed or that crashed processes are operational. Since local failure detectors run independently at each process, one local failure detector may perceive a failure, but other detectors may perceive it at a different time or not at all.

The failure model allows processes to crash, silently halting their execution. Because of the unpredictable delays experienced by the system, it is impossible to use time-outs to accurately detect a process crash.

We assume that a process communicates with its local failure detector through a special receive-only channel on which the local failure detector may place a new list of id's of processes not suspected to have crashed. We call this list the local connectivity view of the process. Each process considers the last local connectivity view received from its local failure detector as the current one.

2.2 Election Specifications

The Election problem is described as follows: At any time, at most one process considers itself the leader, and at any time, if there is no leader, a leader is eventually elected. More formally, the Election Problem is specified by the following two properties:

- *Safety*: All processes in the local connectivity

view of the process never disagree on a *leader*.

- *Liveness*: All processes should eventually progress to be in a state in which all processes connected to the system agree to the *only one* leader.

3. Circumventing The Impossibility Result

In this section, we relate the election specification to other ways to solve the election problem.

- In an asynchronous model augmented by global failure detectors, processes have access to modules that (by definition) eventually reflect the state of the system. Therefore, progress and safety can be guaranteed unconditionally.
- In a timed asynchronous model, processes must react to an input, producing the corresponding output or changing state, within a known time bound. Under this model, progress and safety can be guaranteed if no failures and recoveries occur for a known time needed to communicate in a timely manner.
- In a completely asynchronous model, progress cannot always be guaranteed without violating safety and failure detectors in practice eventually reflect the system state, but they must be considered arbitrary. Correct processes react in practice within finite time, but this time cannot be quantified. Therefore, in order to guarantee a solution, we need a weaker specification of the problem.

Our approach falls into the last category that originated with Garcia-Molina's work [2]. Our election algorithm, however, differs from Garcia-Molina's in several ways.

- Processes in Garcia-Molina's model do not need to wait to get consensus about the current leader's crash. If one process suspects that the leader failed, it may attempt to elect the new leader. Garcia-Molina's specification says that, if one process attempts to be a new leader, it eventually should be elected as a leader. Our specification requires all processes in a set to agree on the current leader crash before changing their new leader.

- Garcia-Molina's specification allows a solution in which the attempted change of a leader divides all processes into several sub-groups. Our specification does not allow such a sub-group because it states that if all processes in a system agree on a new leader, they must eventually accept such a leader.

In our model stability is also required for progress, but, at variance of the above case, it is not necessarily related to the state of the system. In other words, eventual progress is required when there is agreement among a set of the local failure detectors, even if failures and recoveries continue to occur in the system.

4. Election Algorithm

We provide a stable algorithm that solves the Leader Election problem given in Section 2. The algorithm is based on the three asynchronous phases.

- A prepare phase, in which a process propose a new leader that the other processes agree with.
- A ready phase, in which all process that agree on the new leader acknowledge the reservation of the potential leader.
- A commit phase, in which the new leader is finally elected, and all process accept it their only leader.

4.1 Solution Sketch

The main idea for the algorithm is as follows. A process p that is informed by its local failure detector of a leader's crash and that has the smallest id among processes in its new local connectivity view sends a message to all processes in its view proposing to change the current leader with the new leader.

Each process received the message records this proposal until the potential leader in its local view is the same as the proposed new leader in its local view. At which point, it responds by sending back an Accept or Retry message to the process that proposed the leader update. The Accept message is sent if the process agrees on the proposed leader in its local current view.

Upon sending the Accept message, the process reserves the prospective leader, so that no other proposal is accepted for that system. Upon receiving a Retry message, the proposing process returns the normal state of the algorithm, sending a new Abort message to all processes in its view.

When the proposing process has collected Accept messages from all processes in its view, it starts the commit phase by sending commit messages, ordering other processes in its view to commit the leader update. Upon receiving a commit message, the processes accept the reserved prospective leader as a their new leader.

4.2 Code Description

The code is shown in Fig. 1. The first received command in Fig. 1 shows how a process p , when informed of a change in its local connectivity view, set its view to be current and checks if the current leader has crashed. If the leader has crashed, it set the variable *LeaderStatus* to be false. When *LeaderStatus* is false, the *StartElection* procedure in Fig.1 is called and the process p checks that it is the minimum id among the processes in v_p . If p is the minimum id, it increases the round and proposes itself as a new prospective leader and initializes its *ack* array to zero.

The next received commands in Fig. 1 check for incoming messages from other processes. These may be proposals for a new leader (Propose), rejections to propose a new leader (Rejection), acceptances of a proposed new leader (Accept), orders to commit a new leader (Commit) or orders to abort a proposed new leader (Abort).

Upon receiving a proposal message from process q , process p stores the new leader's id proposed by q at position q of the array *NewLeader* and stores the proposed round at position q of the array *RoundIn*, then sets position q of the array *Prop* to true to record the receipt of the proposal from q and sets the *CurView* to false to refresh the current view of the system.

<p>Upon received v_p from FD: <i>CurView</i> := true; If <i>CurLeader</i> $\notin v_p$ then <i>LeaderStatus</i> := 0; end-if If $p = \min(v_p)$ then <i>Round</i> := <i>Round</i> + 1; Call <i>start_election</i>(); end-if</p> <p>Upon received (<i>Propose</i>, <i>PropLeader</i>, <i>k</i>) from <i>q</i>: <i>Prop</i> := true; <i>CurView</i> := false; <i>NewLeader</i> := <i>PropLeader</i> ; <i>RoundIn</i> := <i>k</i>; Call <i>reply_election</i>();</p> <p>Upon received (<i>Reject</i>, <i>k</i>) from <i>q</i>: If <i>Round</i> = <i>k</i> then Send (<i>Abort</i>, <i>Round</i>) to $\forall j \in v_p$; For $\forall j \in v_p$, <i>ack</i>[<i>j</i>] := 0; end-if</p> <p>Upon received (<i>Accept</i>, <i>k</i>) from <i>j</i>: If <i>Round</i> = <i>k</i> then <i>ack</i>[<i>j</i>] := 1; If for $\forall q \in v_p$, <i>ack</i>[<i>q</i>] = 1 then Send (<i>Commit</i>, <i>PropLeader</i>, <i>Round</i>) to $\forall q \in v_p$; For $\forall q \in v_p$, <i>ack</i>[<i>q</i>] := 0; end-if end-if</p>	<p>Upon received (<i>Commit</i>, <i>PropLeader</i>, <i>k</i>) from <i>j</i>: If <i>RoundIn</i> = <i>k</i> then <i>CurLeader</i> := <i>PropLeader</i>; <i>LeaderStatus</i> := 1; end-if</p> <p>Procedure <i>Start_election</i>(): <i>PropLeader</i> := <i>p</i>; Send (<i>Propose</i>, <i>PropLeader</i>, <i>Round</i>) to $\forall q \in v_p$; For $\forall q \in v_p$, <i>ack</i>[<i>q</i>] := 0;</p> <p>Procedure <i>Reply_election</i>(): If (<i>CurView</i> \wedge <i>Prop</i>) then <i>Prop</i> := false; If (<i>NewLeader</i> $\leq \min(v_p) \wedge$ <i>RoundIn</i> $>$ <i>Round</i>) then Send (<i>Accept</i>, <i>PropIn</i>) to <i>q</i>; <i>Next</i> = <i>RoundIn</i> + 1; end-if else Send (<i>Reject</i>, <i>PropIn</i>) to <i>q</i>; end-if</p>
--	---

Fig. 1. The Algorithm.

Upon receiving a proposal message from process *q*, process *p* stores the new leader's id proposed by *q* at position *q* of the array *NewLeader* and stores the proposed round at position *q* of the array *RoundIn*, then sets position *q* of the array *Prop* to true to record the receipt of the proposal from *q* and sets the *CurView* to false to refresh the current view of the system.

If process *p* later agrees on the proposed new leader, it sends a response to process *q* (see last guarded command in Fig. 1). The response is either an acceptance of the new leader at position *NewLeader*[*q*] if the minimum id among the process in v_p is greater or equal than the id of proposed *NewLeader*[*q*] and the proposed round greater than the current round; or it is a rejection to the proposed new leader if the minimum id among the process in v_p is less than the id of

proposed *NewLeader*[*q*] or the proposed round less or equal than the current round.

A rejection to the proposed new leader consists of sending back to *q* the proposed round. An acceptance consists of acknowledging the proposed new leader at position *NewLeader*[*q*]. We now examine the guarded commands of the remaining message types. A process *p* that receives a rejection to the its proposal sends all processes in v_p a message to abort the proposed round and reinitializes the ack array to zero.

A process *p* that receives an acceptance regarding its proposed new leader receives the proposed round. If the received round is equal to the round of the most recent proposal sent, process *p* sets the element at position *q* in the array *ack* to 1 to record the acceptance.

Then, it inspects the ack array to check if all

entries are 1. If so, p starts the commit phase by broadcasting its previously proposed new leader and the corresponding proposed round PropRound to all processes in v_p and reinitializes the ack array to zero.

A process p that receives an order to commit a new leader at position q from process q , simply sets the current leader to the proposed new leader and sets the current round to the proposed round.

5. Correctness

We can ensure the correctness of the algorithm by proving that it satisfies the two properties of the specification given in Section 4.

5.1 Safety

Theorem 1. *The algorithm described in Section 4 satisfies the safety condition of the specification (Property 1, Section 2): At any point in time, all processes connected the system never disagree on a leader.*

Proof. Either all processes remain in the start state or some process p receives the proposed leader as its leader. In the start state, the safety property holds since all processes are in the state in which a leader has not been elected. If some process p receives its leader by committing a proposed leader at a given position q , it must have received a Commit message from some process q ; therefore, q must have received Accept messages regarding its proposal of a new leader from all processes in v_p including p . It follows from the last guarded command in Fig. 1 that, if process p has accepted the proposal of process q , it will not accept any other proposal for new leader, making it possible to commit at most single proposed leader. Therefore, process p either commits the process at position q as a new leader or ends up with position q by aborting the proposed new leader. Therefore safety property holds. \square

5.2 Liveness

Theorem 2. *The algorithm described in Section 4 satisfies the liveness condition of the specification (Property 2, Section 4): All processes should eventually progress to be in a state in which all processes connected to the system agree to the only one leader.*

Proof. By contradiction, a non-progress means that the new leader is not elected forever even though there is no leader; therefore, no Commit messages must be sent. Since the number of processes is finite, there must be at least one process whose id is the minimum value in v_p and that process eventually sends a Propose message. Call this process p . By the code in Fig. 1, we see that, to have no Commit message, each time p sends a Propose message, it should be rejected by other process. It follows that, in order to abort infinitely many Propose messages, other process q must reject the proposed messages infinitely often. Propose messages are rejected either when the minimum id of v_p is greater than the id of the proposed leader or because of a Propose message already has been received (see Fig. 1).

The first case is ruled out because it implies that some process always considers that there is a process that is alive and whose id is less than the id of proposed new leader. But by strong completeness of a failure detector it is contradiction.

The second case is also ruled out, because it implies that other process q sends infinitely many proposals of the other leader. But by eventual strong accuracy of a failure detector, the process q knows that there is a process whose id is less than its id. Therefore it is contradiction. \square

6. Concluding Remarks

We have presented a stable election protocol with a reliable failure detector in completely asynchronous systems. We have assumed our local failure detectors to be inaccurate and incomplete. With this approach, the leader election specification states explicitly that progress without violation of safety cannot always be guaranteed. In practice, our requirement for progress is weaker than that stated in the original specification of having a set of processes sharing the same leader.

In fact, if the rate of perceived a leader failures in the system is lower than the time it takes the protocol to make progress and accept a new leader, then it is possible for the algorithm to make progress every time there is a leader failure in the system. This depends on the actual rate of a leader failures and on the capacity of the failure detectors to track such failures.

In [10], Chandra and Toueg note that failure detectors defined in terms of global system properties cannot be implemented. This result gives strength to the approach of relaxing the specification and of having a stable election protocol. In real world systems, where process crashes actually lead a connected cluster of processes to share the same connectivity view of the network, convergence on a new leader can be easily reached in practice.

References

- [1] G. LeLann, "Distributed Systems—towards a Formal Approach," in *Information Processing 77*, B. Gilchrist, Ed. North-Holland, 1977.
- [2] H.Garcia-Molian, "Elections in a Distributed Computing System," *IEEE Transactions on Computers*, vol. C-31, no. 1, pp. 49-59, Jan. 1982.
- [3] H. Abu-Amara and J. Lokre, "Election in Asynchronous Complete Networks with Intermittent Link Failures." *IEEE Transactions on Computers*, vol. 43, no. 7, pp.778-788, 1994.
- [4] H.M. Sayeed, M. Abu-Amara, and H. Abu-Avara, "Optimal Asynchronous Agreement and Leader Election Algorithm for Complete Networks with Byzantine Faulty Links.," *Distributed Computing*, vol. 9, no. 3, pp.147-156, 1995.
- [5] J. Brunekreef, J.-P. Katoen, R. Koymans, and S. Mauw, "Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks," *Distributed Computing*, vol. 9, no. 4, pp.157-171, 1996.
- [6] G. Singh, "Leader Election in the Presence of Link Failures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 3, pp.231-236, March 1996.
- [7] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, pp. 374-382. (32) 1985.
- [8] T. Chandra and S.Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *Journal of ACM*, Vol.43 No.2, pp. 225-267, 1996.
- [9] D. Dolev and R Strong, "A Simple Model For Agreement in Distributed Systems," In *Fault-Tolerant Distributed Computing*, pp. 42-50. B. Simons and A. Spector ed, Springer Verlag (LNCS 448), 1987.
- [10] T. Chandra, V. Hadzilacos and S. Toueg, "The Weakest Failure Detector for Solving Consensus," *Journal of ACM*, Vol.43 No.4, pp. 685-722, 1996.

Genetic Ensemble (G-Ensemble) for Meteorological Prediction Enhancement

Hisham Ihshaish*, Ana Cortés, and Miquel A. Senar
Department of Computer Architecture and Operating Systems,
Universitat Autònoma de Barcelona, Barcelona, Spain
hisham@caos.uab.es, ana.cortes@uab.es, miquelangel.senar@uab.es

Abstract—*The need for reliable predictions in environmental modelling is long known. Particularly, the predicted weather and meteorological information about the future atmospheric state is crucial and necessary for almost all other areas of environmental modelling. Additionally, right decisions to prevent damages and save lives could be taken depending on a reliable meteorological prediction process. Lack and uncertainty of input data and parameters constitute the main source of errors for most of these models. In recent years, evolutionary optimisation methods have become popular to solve the input parameter problem of environmental models. We propose a new prediction scheme that uses a Genetic Algorithm for parameter estimation in Numerical Weather Prediction Models (NWP) to enhance prediction results. The new approach is called Genetic Ensemble (G-Ensemble) and it has been tested using historical data of a well known weather catastrophe: Hurricane Katrina that occurred in 2005 in the Gulf of Mexico. Obtained results provide significant improvements in weather prediction.*

Keywords: numerical weather prediction; evolutionary computing; genetic algorithm; ensemble prediction; parameter estimation.

1. Introduction

Weather forecasting and prediction is an ongoing demand since thousands of years. Agriculture, education, entertainment, industry, astronomy, etc. usually benefit from an accurate knowledge of the weather future state. Global weather predictions are held by governments and international scientific institutions, to provide information about the present and time evolution of the atmospheric situation. However, regional predictions in certain zones are done by local organizations, governments, and scientific centers to provide predictions on basis of fine-coarse resolutions.

Weather time evolution is represented by numerical models that are commonly solved by means of computing facilities. Efforts initiated in the 1950s when the USA National Weather Service (NWS) [1] began to utilize some of the early versions of computers to make large-scale

weather forecasts. Since that time, computers have become faster and more sophisticated being able to provide the scientific community (particularly to the weather forecasting community) with High Performance Computing platforms, which allow the execution of highly computing demanding weather forecast simulations. However, scientific applications continue to be more complex while research is getting more sophisticated as a result of the natural human growth of requirements. Higher accuracy, larger time scales, more complex problems and less waiting time constitute some of the new demands that should be considered from a computational point of view.

Numerical Weather Prediction (NWP) models are considered as soft-real time applications. The importance of having a degree of accuracy in the prediction in a certain time is a real challenge. Thus, ongoing investigations concentrate on methods to enhance the process of prediction, and to get results of this process faster.

As most simulation software works with well-founded and widely accepted models, the need for input parameter optimisation to improve model output is a long-known and often-tackled problem. Particularly in environments where correct and timely input parameters cannot be provided, efficient computational parameter estimation and optimisation strategies are required to minimise the deviation between the predicted scenario and the real phenomenon behaviour. With the continuously increasing availability of computing power, evolutionary optimisation methods, especially Genetic Algorithms (GA), have become more popular and practicable to solve the parameter problem of environmental models.

This work presents a new meteorological prediction scheme that uses evolutionary optimization methods that enhance the quality of weather forecast by focusing on the calibration of input parameters.

The rest of the paper is organized as follows: Section 2 gives an overview of NWP models with 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 focuses on the importance of accuracy in NWP models and it describes also the most widely used methods for NWP enhancement in practise. In section 4, the proposed prediction scheme (*G-Ensemble*) is presented and described. Section 5 presents

This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974.

*Corresponding author.

†This paper is addressed to the PDPTA conference.

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 and WRF

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 treats 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.

Certain areas where atmosphere future conditions are to be predicted are represented by three dimensional uniform-gridded-rectangles referred as domains. The input data which corresponds to the actual state of the atmosphere is called initial conditions. Those initial conditions are assigned to all points of the grid. The horizontal distance between grid points is referred as the 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 cannot be represented on the coarser grid of a global model. Hence, a NWP model will guess the new values of the initial conditions over future time scale.

The Weather Research and Forecasting (WRF) [2] is a widely-used numerical weather prediction model, which is considered as a next-generation mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. WRF is composed of a variety of programs to facilitate prediction process, such as extracting global terrain data, designing domains, facilitating for real observations to be injected while model integration, and post-processing outputs.

In this work, we developed a new methodology for meteorological prediction enhancement using WRF as the Numerical Weather Prediction model. 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

Reliable weather predictions may not prevent disasters, but at least they help in preventing their horrible effects, such as reducing the possibility of large property damages and even could help in saving lives. Furthermore, accurate

predicted meteorological variables are critically needed for other environmental modelling systems. For example, 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. It is clear that, in such cases, accurate predictions may contribute also to save human lives. Air pollution modelling and the short behaviour of natural disasters like hurricanes are other examples where reliable predictions of meteorological variables are also necessary.

The importance of reliable weather predictions motivated relevant improvements of NWP in the last 30 years. Efforts have been done in the field to enhance predictions [4], however, many sources of errors still remain. The main ones are the availability and accuracy of input data (initial conditions) on higher resolution basis, the possibility of data injection of real observations during prediction process and physical parametrization.

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 less than 1 kilometre, while NWP models predicts normally on domains of grid-scales higher than 1 kilometre. Parametrization is needed in such cases to represent this process on a certain domain scale. Other examples are the typical cumulus cloud which has a scale of less than 1 kilometre, the amount of solar radiation that reaches the ground, and interactions with the surface, including the generation of drag and waves by orography. And so, all of these processes must be parametrized before they can be included in the model.

Summarizing, there is an important need to get reliable weather predictions, while it is also known that the major sources of error that reduce prediction accuracy are input data [5], availability of observed data, and the parametrization process. Thus, the efforts to enhance NWP are mainly focusing in enhancing input data, enabling injection of observed data, and estimating correctly the parameters of sub-scale parametrization process.

The two mostly used NWP enhancement methods are Three-Dimensional Variational Data Assimilation (3DVAR) and Ensemble Prediction System (EPS), which are still a center of continuous research. Actually, both methods fall within the general approach of Data Assimilation (DA) [6] for numerical prediction models. A Data Assimilation system combines all available information about atmospheric

state to produce an estimate of initial conditions valid at a prescribed analysis time. It proceeds by analysis cycles. In each analysis cycle, observations of the current (and possibly, past) state of the atmosphere are combined with the results from a NWP model (the forecast) to produce an analysis, which is considered as "the best" estimate of the initial conditions of the system. DA tries to balance the uncertainty in input data and in the forecast. The model is then advanced in time and its result becomes the forecast in the next analysis cycle.

Next, we describe in more detail the two approaches that are most widely used for NWP enhancement.

3.1 Three-Dimensional Variational Data Assimilation

3DVAR [7] uses information which include observations, previous forecasts (background or first-guess), their respective errors and the laws of Physics to produce the analysis.

The basic goal of the 3DVAR system is to produce an "optimal" estimation of the true atmospheric state at analysis time, which is achieved by finding an iterative solution of a prescribed cost function, described in detail in [7]. This solution represents a minimum variance estimate of the true atmospheric state having two sources of data: background (previous forecast) and observations. This process includes the implementation of certain algorithms to estimate background, and observation errors.

The main drawback of this method consists of the necessity of roll-back the simulation process in order to inject the new data in such a way that corrects the observed error in a progressive way as simulations go on. This way of working increases the execution time of the prediction process due to the need of re-starting the model execution from scratch.

3.2 Ensemble Prediction System (EPS)

Stochastic or "ensemble" forecasting is used to account for uncertainty. It involves multiple forecasts created with an individual forecast model by using different physical parametrizations or varying initial conditions. The ensemble forecast is usually evaluated in terms of an average of the individual forecasts concerning one forecast variable, as well as the degree of agreement between various forecasts within the ensemble system, as represented by their overall spread [8], [9]. In [10] they show how NWP models are sensitive to the choice of physical parametrization and how an ensemble could be established using these parametrizations.

A set of forecasts is then produced (each of which has a different set of initial conditions or a different physical parametrization) using a deterministic model to predict the future state of the atmosphere, and by assuming that the model is perfect without other errors, then the mean of all of the executed simulations (forecasts) is considered to be the true future state of the atmosphere.

The implementation of this method begins with the process of determining how to select the set of the various initial conditions or parametrizations as presented in [11]. As soon as this set is established, the corresponding simulations are executed to predict the relative evolution of atmospheric fields in the short future time.

EPS could be considered as a parallel method as each ensemble member is actually a stand-alone simulation, which can be executed independently of the others. Therefore, the main drawback of this scheme is the need of a huge computing power to be able to run all simulations in parallel.

4. Genetic Ensemble (G-Ensemble)

In this section, the Genetic Ensemble (*G-Ensemble*) approach for prediction enhancement is described. Although *G-Ensemble* uses the same principles of the EPS, it clearly differs in the way of how ensemble members are obtained and executed. The main idea of an EPS is to reflect possible variations in the ranges of some input parameters, thus, they simply run a variety of predictions, each of which is initiated with a different combination of those input parameters. Then, the average of all predictions results is considered as the best prediction as it actually reflects a range of variations in certain input parameters. We propose a new scheme of prediction, shown in figure (1) where we introduce a pre-prediction phase or stage, called Calibration Phase, which ends at the moment where real observations are available. Hence, the whole prediction process will be formed of two stages: Calibration and Prediction, which we describe below.

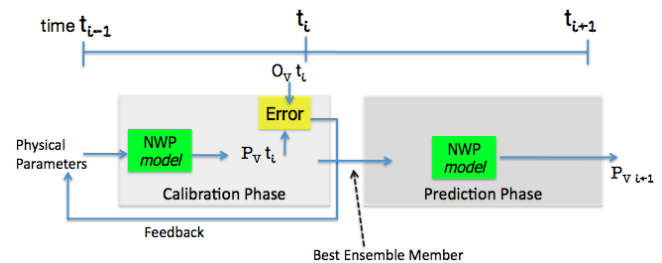


Fig. 1: Two-phase prediction scheme; t_i is time 00:00 of prediction process, t_{i-1} is a time instant previous to Prediction Phase (initial time of Calibration Phase), t_{i+1} is the future time to be predicted. " O_V " is an observed meteorological variable at time t_i , " P_V " is the predicted variable at the same time using a NWP model.

4.1 Calibration Phase

Considering that t_i is the instant time from which the meteorological variables are going to be predicted, Calibration Phase starts at a time prior to prediction time and ends at time 00:00 of prediction period, i.e. calibration is done within the period (t_{i-1}, t_i) . Knowing that real observations of meteorological variables are available at time t_i , the objective of this phase is to look for the combination of

the physical parameters, which produce less error compared to these observed meteorological variables at the end of the phase (at time t_i). That is, as in EPS, we initialize a set of simulations randomly, each of which has a different physical parametrization combination. This initial set, which we call *initial ensemble*, is run by the NWP model to predict meteorological variables at time t_i , then we use GA functions to obtain an improved ensemble set (which has less errors compared to observations at time t_i) and the process is repeated again many times to a certain number of iterations. At the last iteration of the GA, Calibration Phase exits with calibrated ensemble members that we refer as *G-Ensemble*, each of which has a calibrated combination of physical parameters, which produced less error than those of the *initial ensemble*. At that point, we have two alternatives for the Prediction Phase: 1) to apply the classical EPS scheme using the obtained *G-Ensemble* set, or 2) to select the ensemble member of the *G-Ensemble* with minimum error, to be the single ensemble member of the simulation that will conduct the Prediction Phase. We call this approach *Best Genetic Ensemble Member (BeGEM)*.

A relevant point to be considered in the Calibration Phase is the error definition being one of the core elements of this phase. In this work, we propose two different error functions to be used, what we call Single-Variable and Multi-Variable. Depending on the error function used, we have designed two *G-Ensemble* strategies: *Single-Variable G-Ensemble* and *Multi-Variable G-Ensemble*, which are described below.

4.1.1 Single-Variable G-Ensemble

The Calibration Phase is done with the goal of enhancing predictions for a single meteorological variable. The error function for the evaluation of ensemble members in our GA is the Root Mean Square Deviation *RMSD* or Error *RMSE*, shown below in equation(1). This error function is a frequently-used measure for the evaluation of meteorological predictions [12], which measures the differences between values predicted by a model or an estimator and the values actually observed from the variable being estimated. In *RMSD* equation, x_{obs} is an observed value of a variable x and x_{pre} is the predicted one for the same variable.

$$RMSD = \sqrt{\frac{\sum_{i=1}^n (x_{obs,i} - x_{pre,i})^2}{n}} \quad (1)$$

Using *RMSD* error in the Calibration Phase limits our *G-Ensemble* to be oriented to enhance predictions for one meteorological variable at a time. For example, we can use it to enhance predictions of Temperature or Precipitation, but not for both at the same time. This occurs because the error used produces a value of the variable unit that can not be compared with other variables. In order to overcome such a

drawback, we proposed an alternative error function, which we refer as *Multi-Variable G-Ensemble*.

4.1.2 Multi-Variable G-Ensemble

The calibration is done with the goal of enhancing the prediction of multiple meteorological variables at the same time. To bypass the limitation imposed by *RMSD* error, we use the Normalized *RMSD*, see equation (2).

$$NRMSD = \frac{\sqrt{\frac{\sum_{i=1}^n (x_{obs,i} - x_{pre,i})^2}{n}}}{x_{obs(max)} - x_{obs(min)}} \quad (2)$$

The Normalized *RMSD* (referred as *NRMSD*) is the value of *RMSD* divided by the range of the observed values of a certain variable. *NRMSD* indicates the error percentage of the predicted value of a certain variable, compared to its observed values. In order to consider more than one variable at a time, we evaluate *NRMSD* for all variables, and then, we consider the addition of all of them as the Multi-Variable error function. For example, the *NRMSD* of a model that predicts Temperature (T) and Precipitation (P) is the percentage obtained by the summation of two Percentages: $NRMSD(T)$ and $NRMSD(P)$, as shown in equation (3).

$$Error = NRMSD(var1) + NRMSD(var2) = value\% \quad (3)$$

Therefore, the Calibration Phase and, particularly the GA, considers this error function as the objective function used to sort the intermediate individuals of the ensembles.

4.2 Prediction Phase

Once the Calibration Phase is finished, it is the turn of the Prediction Phase. At this point, either the *BeGEM* or the whole *G-Ensemble* set produced by the previous phase will be run by the NWP model. It is expected that this ensemble member will generate better predictions as it shown less error in Calibration Phase. In contrast to the classical EPS, only one simulation is executed here, while in EPS the whole ensemble set is executed.

5. Experimental Test Case

To test our approach, we used historical data of hurricane Katrina [13], see a picture in figure (2). Katrina 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).

To Predict meteorological variables, we used WRF as the NWP model and, we used the coupled NOAA Land Surface Model (NOAH LSM) [14] for land surface physical parametrization. At runtime, NOAA LSM provides important values to WRF that correspond to subgrid-scale evolution of land surface variables (surface sensible heat flux,



Fig. 2: Satellite picture of hurricane Katrina on Aug. 29, 2005 at 12:15 p.m

surface latent heat flux, skin temperature, surface emissivity and the reflected short-wave radiation). It calculates these variables depending on a set of parameters that characterize the land surface: *Landuse* and *Soil* parameters [15]. As a result, predictions are enhanced when LSM is used as more subgrid-scale meteorological variables are injected into the model. However, these parameters fall within ranges and small changes in their values produce non-negligible differences in prediction results. The EPS comes at this point to solve the problem by generating a number of predictions, each of which has different values of *Landuse* and *Soil* parameters, hence, the final result of the prediction will be the average of the results of all predictions which are supposed to cover an "acceptable" variation in physical parametrization (land surface parametrization).

The objective of the experiments is to predict meteorological variables evolution 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 evolution of meteorological variables is produced every 3 hours.

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 variables predicted in our experiments were: *Latent Heat Flux LHF (W/m²)*, *Surface Skin Temperature TSK (K)*, *2-meter Temperature (K)*, *10-meter Wind Velocity components U10 and V10 (m/s)*, and the *Accumulated Precipitation RAINC (mm)*.

In the next two subsections, we discuss results by which we make a comparison between classical EPS and the *G-Ensemble*. Furthermore, we also analyse the computational cost incurred by both approaches.

5.1 Ensemble Vs. G-Ensemble

In this section, a comparison of prediction results is done between the classical EPS and our method (*G-Ensemble*). Figure (3) shows an experiment result of using classical EPS of 40 ensemble members (each of which has a different *Landuse* and *Soil* parameters) to predict *Latent Heat Flux LHF* variable. As shown in the figure, each line represents the predicted values of LHF every 3 hours. The dotted line represents the average of all of those predicted values of all simulations, which will be considered as the best prediction result according to the classical EPS.

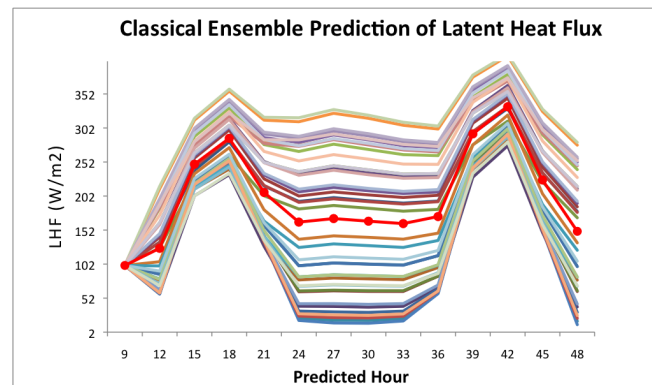


Fig. 3: Classical Ensemble of size:40 to predict Latent Heat Flux LHF.

We applied our method with *Single-Variable G-Ensemble* in two different cases: to predict LHF (results shown in figure 4) and to predict Acc. Precipitation (results shown in figure 5). In both cases, with the same initial ensemble members, we obtained a significant improvement in prediction quality. The Genetic Algorithm of the Calibration Phase was configured to iterate 20 times over an initial population size of 40 individuals (initial ensemble size). Its three main operators were configured as follows: *Selection*: (best one of two) and (roulette), *Crossover*: (probability=0.7, type: two points crossover), and *Mutation*: (probability= 0.2).

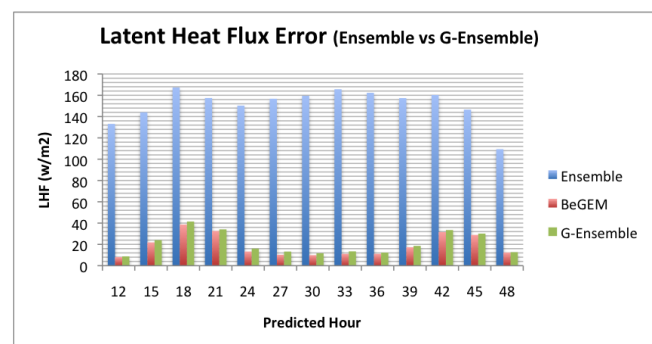


Fig. 4: Single-Variable *G-Ensemble*; *RMSD* error in prediction of variable LHF.

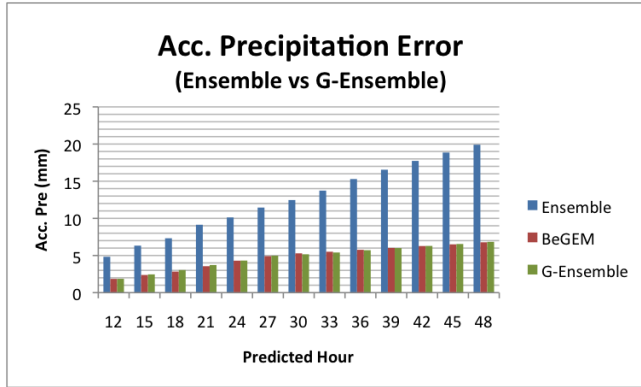


Fig. 5: Single-Variable *G-Ensemble*; *RMSD* error in prediction of variable *Acc. Precipitation*.

The results of the Calibration Phase are the enhanced 40 individuals (*G-Ensemble* members). As shown in figures 4 and 5, the average error of *G-Ensemble* predictions is always less than the average error of the classical EPS referred as Ensemble in the figures. Furthermore, if we just run a single prediction with *BeGEM* of the Calibration Phase, errors are even reduced more.

We also used our approach to enhance predictions of a set of meteorological variables at the same time, by applying the *Multi-Variable G-Ensemble* and using the error *NRMSD* (shown in equations 2 and 3) in Calibration Phase as the fitness function of the GA. In this case, we were also able to obtain significant improvements in the prediction of a set of meteorological variables at the same time.

Figure 6 shows the results obtained in this case. Again, significant reduction of the *NRMSD* were obtained in the prediction of a set of meteorological variables together.

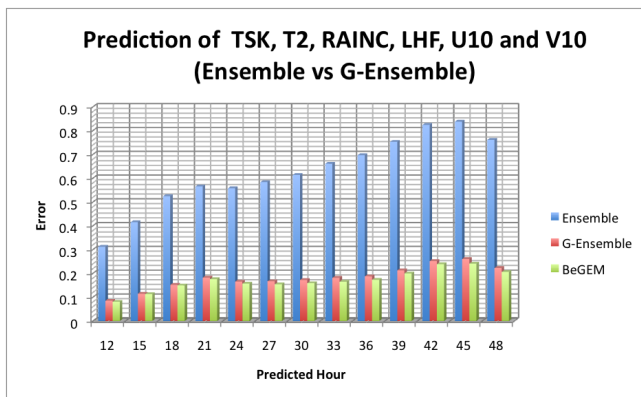


Fig. 6: *Multi-Variable G-Ensemble*; *NRMSD* in prediction of variables: *Latent Heat Flux LHF*, *Surface Skin Temperature TSK*, *2-meter Temperature*, *10-meter Wind Velocity components U10 and V10*, and the *Accumulated Precipitation RAINC*.

Additionally, we observed that a reduction of the *NRMSD* of a set of variables also provides an enhancement

in the prediction of each meteorological variable alone. In other words, all six variables were better predicted when *G-Ensemble* oriented to reduce the *NRMSD* of those variables together. To illustrate these results, we show in figure (7) how the corresponding prediction error of *Latent Heat Flux LHF* was reduced by the *G-Ensemble* oriented to reduce the *NRMSD* of the six variables (the same effect was observed in the other five variables).

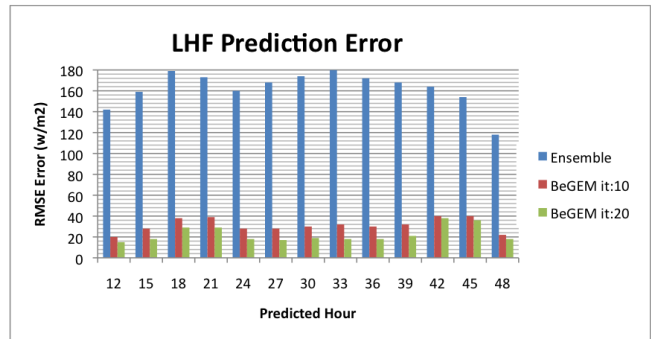


Fig. 7: *RMSD* prediction error of *Latent Heat Flux LHF*(*W/m2*) in prediction using *BeGEM* produced in iterations 10 and 20 of the Calibration Phase of the *Multi-Variable G-Ensemble*.

5.2 Accuracy versus Cost

The problem of the uncertainty in NWP initial conditions produces what is called "imperfectness" in prediction accuracy. The previous mentioned methods, among others [16]–[18], are implemented to reduce the margin of the "imperfectness" in prediction accuracy. However, the trade-off between cost (execution time) and prediction accuracy is an important factor that should be considered to select the most suitable enhancement method.

In scenarios with a limited number of computational resources, EPS is not an eligible method as it needs lots of resources to execute a set of predictions. Using our approach, we obtained a significant reduction of computational time when we executed an experiment comparing classical ensemble with *G-Ensemble*. Predictions were executed in parallel over a cluster of 80 computing nodes. Figure (8) shows the prediction error of an experiment to enhance prediction of 6 meteorological variables, using classical EPS and the *Multi-Variable G-Ensemble* in 5 different scenarios, which correspond to different GA settings. The execution time of all scenarios and their settings are listed in (table 1).

In four scenarios of *G-Ensemble* (scenarios 2, 3, 4, and 6), we observed a significant reduction in execution time along with its corresponding reduction of prediction error. A classical EPS of 40 ensemble members *Ensemble(40)* could be replaced by any scenario of *BeGEM(40)* calibrated by (5, 10, or 15) iterations of the GA. Similarly, *BeGEM(20)* with 20 initial ensemble members iterated 20 times at

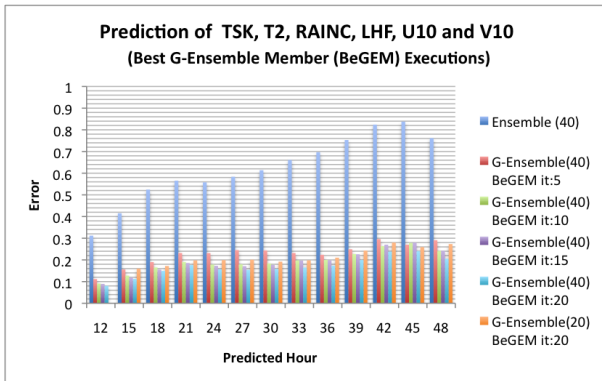


Fig. 8: Multi-Variable *G-Ensemble*; *NRMSE* of prediction of variables: Latent Heat Flux *LHF*, Surface Skin Temperature *TSK*, 2-meter Temperature, 10-meter Wind Velocity components *U10* and *V10*, and the Accumulated Precipitation *RAINC*.

Table 1: Execution time Vs Scenario

Number	Scenario	G-Ensemble	# of Iterations	Ex.Time
1	<i>Ensemble(40)</i>	No	-	1120 m.
2	<i>BeGEM(40)</i>	Yes	5	369 m.
3	<i>BeGEM(40)</i>	Yes	10	709 m.
4	<i>BeGEM(40)</i>	Yes	15	1024 m.
5	<i>BeGEM(40)</i>	Yes	20	1549 m.
6	<i>BeGEM(20)</i>	Yes	20	709 m.

Calibration Phase constitutes another scenario that reduces both prediction error and execution time. Only in one case (scenario 5), our method incurred in an execution time larger than classical ensemble. This is due to the number of iterations used in the Calibration Phase by the GA, which was 20 iterations. Fortunately, significant improvement in prediction quality (almost similar) is gained by calibrating with less number of iterations (as in scenarios 2, 3, and 4), or by reducing the size of initial ensemble members (as in scenario 6). This means that this case could be prevented by either calibrating with less number of iterations or by reducing the size of initial ensemble members.

In summary, *G-Ensemble* method provides the possibility to select between various scenarios considering a balance between prediction quality and prediction cost.

6. Conclusions and future work

In this work, we have briefly described Numerical Weather Prediction models, along with a description of WRF as one of the most widely used models in the field. We highlighted the importance of the accuracy in NWP models, discussing also the basic two methods used for prediction enhancement. We analysed the penalties incurred by these methods in terms of time execution costs and prediction accuracy.

We have introduced *G-Ensemble*, as a new scheme that enhances weather predictions. It uses an evolutionary algorithm to estimate best possible physical parameters that will provide more reliable predictions.

The *G-Ensemble* prediction scheme showed a significant improvement in prediction quality. Thanks to the enhancement in prediction accuracy, more sophisticated schemes might be developed in the near future by injecting observed meteorological variables at run-time. These results encourage us to continue our research efforts by adding methods that handle real observations and deciding their injection intervals at run-time in order to get more reliable meteorological predictions.

References

- [1] National Weather Service (NWS) homepage. [Online]. Available: <http://www.nws.noaa.gov/>
- [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] W. D. Bonner, "NMC overview: Recent progress and future plans," *Weather and Forecasting*, vol. 4, no. 3, pp. 275–285, 1989.
- [5] D. J. Stensrud and J. M. Fritsch, "Mesoscale convective systems in weakly forced large-scale environments. Part II: Generation of a mesoscale initial condition," *Monthly Weather Review*, vol. 122, no. 9, pp. 2068–2083, 1994.
- [6] B. Wang, X. Zou, and J. Zhu, "Data assimilation and its applications," *Proceedings of the National Academy of Sciences*, vol. 97, no. 21, pp. 11 143–11 144, 2000. [Online]. Available: <http://www.pnas.org/content/97/21/11143.abstract>
- [7] D. M. Barker, W. Huang, Y. R. Guo, A. J. Bourgeois, and Q. N. Xiao, "A three-dimensional variational data assimilation system for MM5: Implementation and initial results," *Monthly Weather Review*, vol. 132, no. 4, pp. 897–914, 2004.
- [8] 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.
- [9] T.-Y. Lee and S.-Y. Hong, "Physical parameterization in next-generation nwp models," *Bulletin of the American Meteorological Society*, vol. 86, no. 11, pp. 1615–1618, 2005. [Online]. Available: <http://journals.ametsoc.org/doi/abs/10.1175/BAMS-86-11-1615>
- [10] N. K. Awan, H. Truhetz, and A. Gobiet, "Parameterization induced error-characteristics of mm5 and wrf operated in climate mode over the alpine region: An ensemble based analysis," *Journal of Climate*, vol. 0, no. 0, p. null, 2011. [Online]. Available: <http://journals.ametsoc.org/doi/abs/10.1175/2011JCLI3674.1>
- [11] J. L. Anderson, "Selection of initial conditions for ensemble forecasts in a simple perfect model framework," *Journal of the Atmospheric Sciences*, vol. 53, no. 1, pp. 22–36, 1996.
- [12] C. J. Willmott and K. Matsuura, "On the use of dimensioned measures of error to evaluate the performance of spatial interpolators," *International Journal of Geographical Information Science*, vol. 20, no. 1, pp. 89–102, January 2006.
- [13] Hurricane Katrina homepage. [Online]. Available: <http://www.katrina.noaa.gov/>
- [14] The Community Noah Land-Surface Model homepage. [Online]. Available: http://gcmd.nasa.gov/records/NOAA_NOAH.html
- [15] Research Applications Laboratory (land-surface modeling) homepage. [Online]. Available: <http://www.ral.ucar.edu/research/land/technology/lsm.php>
- [16] D. Zupanski, M. Zupanski, E. Rogers, D. F. Parrish, and G. J. DiMego, "Fine-resolution 4dvar data assimilation for the great plains tornado outbreak of 3 may 1999," *Weather and Forecasting*, vol. 17, no. 3, pp. 506–525, 2002.
- [17] G. Evensen, "The ensemble kalman filter: theoretical formulation and practical implementation," *Ocean Dynamics*, vol. 53, no. 4, pp. 343–367, 2003.
- [18] P. L. Houtekamer, H. L. Mitchell, G. Pellerin, M. Buehner, M. Charon, L. Spacek, and B. Hansen, "Atmospheric data assimilation with an ensemble kalman filter: Results with real observations," *Monthly Weather Review*, vol. 133, no. 3, pp. 604–620, 2003.

Study of mobile collaborative information system using distributed database architecture

Mahmoud Abaza, Duane Cato
ATHABASCA UNIVERSITY

Abstract - Please consider these Instructions as guidelines for preparation of Final Camera-ready Papers. The Camera-Ready Papers would be acceptable as long as it is formatted reasonably close to the format being suggested here. Note that these instructions are reasonably comparable to the standard IEEE typesetting format. Type the abstract (100 words minimum and 150 words maximum) using *Italic font with point size 10*. The abstract is an essential part of the paper. Use short, direct, and complete sentences. It should be brief and as concise as possible.

Keywords: Distributed databases, mobile computing, J2ME, collaborative computing.

1 Introduction

The specific field of mobile distributed database management continues to be a burgeoning area for research, due in large part to the very rapid changes that have occurred in mobile device capability over the past few years. Mobile devices now are capable of rapid processing, high-speed communication and support high-level programming primitives (e.g., using Java Micro Edition - J2ME) and are thus perfect candidates for empowering the average user with improved data accessibility and collaboration, within a mobile context. This can potentially provide benefits in increased usefulness, timeliness, and availability of on-time, real-time information to the everyday mobile user. This paper attempts to provide additional information and context for the inevitable discussions which will be necessary to fully utilize and monetize solutions based upon this technology, through two broad approaches:

- Comparison of Mobile Distributed Databases, and
- Prototype Mobile Distributed Database (MDD) Groupware Solution

The existing MDD comparison solutions report provides details of characteristics and metrics for a number of identified candidate products. These dimensions include:

- feature-set,
- availability,
- operating environment/platform, and
- cost and usability .[2]

Prototype MDD Groupware Solution

The proposed groupware system focuses on distribution and sharing of questions, answers and comments between

members of a 'study-group', utilizing mobile devices to handle the tasks of posting, updating and most importantly, storing communication between group constituents. The design is unusual in that it includes no centralized storage database envisaged in the architecture of this particular solution; all persistent and session data generated and utilized in the course of operation exists as the sum total of information within component participating mobile database nodes [3]. Future enhancement directions may include the possibility of implementing some sort of offload or external backup mechanism, in order to provide long-term persistent storage or archival capability.

It should also be noted that, as a prototype, the primary goal of this solution was to illustrate existing design and operational morphologies of the specific MDD identified from the evaluation phase of the project, as well as potentially identify improvements and innovations in existing infrastructure and design, that could lead to performance, reliability or functional improvements in the MDD arena.

These factors all impact the deliverability and usability of the solution in developing, deploying and operating solutions based on the particular MDD technology

An electronic copy of your *full camera-ready paper* must be uploaded (in PDF format) to Publication Web site before the announced deadline. Please follow the submission instructions shown on the web site. The URL to the website is included in the notification of acceptance that has been emailed to you by Prof. Arabnia.

2 Definitions

To satisfy the objectives of this study, it was necessary to identify the resource costs related to the management of data objects across the distributed nodes of the database. These costs can be segregated into object management costs, communications costs and I/O costs. For our purposes, the approach outlined by Huang and Wolfson, [7] was sufficient, particularly due to its specificity for determination of object allocation and access costs. Their method analyzes the cost of distributed object management algorithms in stationary and mobile computing environments. As a precursor to the discussion of their methodology, we include a few definitions:

- A distributed database can be defined as a database that is not stored at any one single physical location, but rather is

dispersed across a network of interconnected computers or devices

- An execution schedule is a sequence of requests, with its own associated execution set (reads and writes).
- A saving-read is a read operation that results in saving the data object locally rather than to a remote node
- An allocation schedule is an execution schedule where some of the read requests are saving-reads. At the end of the allocation schedule the data object is stored in the local databases of the participating nodes.
- A legal allocation schedule is one in which the execution set for every read request contains a reference to a valid (in the network) node or processor; i.e., a node with the latest version of the data object in its local database.
- Allocation scheme for a request is the set of nodes (or processors) that have the latest version of the data object in the local database, just before execution.
- A distributed object management algorithm (DOM) is defined as an algorithm which generates a legal allocation schedule based on an initial allocation schedule.

3 Methodology

This project examines the feasibility of communal information sharing between mobile devices using a distributed architecture for the underlying database topology. The first objective, examination and review of available technologies and products currently supporting distributed mobile database functionality, was achieved through review of public benchmark information for a number of popular mobile distributed database (MDD) products. The second objective was met through development of a prototype mobile database application using a distributed database architecture to implement information management functions implementing one of the MDD candidates identified above, in combination with development of any required extensions or custom distributed data management functionality not already provided by the chosen MDD infrastructure. The following details the rationale behind the approach and evaluation techniques followed in this research.

Distributed Database Infrastructure

In order to satisfy the unique needs of data management in a distributed environment, distributed information retrieval (DIR) techniques are more appropriate than the centralized methods common to monolithic stand-alone databases. In a DIR, all participating nodes in the distributed environment are indexed, to identify those that are likely candidates for locating the particular information desired; only those that meet the search criteria are included in a final list of search hosts. The assumption here is that each participating node in the database, indexes its own subset of data, and thus can answer the question of what information is contained therein. Also, this mechanism presupposes the availability of all the

hosts in the database: disconnected of nodes in the database will lead to skewed, or even incorrect search results.

The improvement of the underlying fault-tolerance of the databases' network connectivity will lead to a concomitant increase in the reliability and accuracy of search and data management operations from the overall information management system. Mechanisms for increasing the availability and recoverability within a mobile distributed context are limited by a variety of operational parameters (e.g., infrastructure cost, data transmission cost, network latency, bandwidth, underlying connection protocol artifacts). For purposes of this analysis, we focus principally on reducing data management and transmission costs, through the use of enhanced data transmission protocols and node selection schemes. Candidate methodologies for DIR node interaction include:

- i. Communication through a centralized server, which manages the process of data synchronization between the nodes in the mobile distributed database. Issues related to this method include synchronization and federation update consistency, as well as performance bottlenecks and single-point failure concerns.
- ii. Communication in an ad hoc manner as necessary for synchronization between individual nodes in the distributed database. This remedies the single-point failure issues identified in (i), but only changes the nature and cause of performance and synchronization concerns.
- iii. Communication between peers in the distributed database, using an enhanced protocol and associated topology to avoid failure sensitivity and performance issues associated with options (i) and (ii) above. Synchronization issues continue to require creative management, particularly in light of the more complex interaction now occurring between peers.

Validation of the approaches indicated above, requires a quantitative determination of cost for implementation, management and system resource utilization. Assuming that external costs of management and implementation remain consistent between the three options, the varying cost becomes that required for ongoing system resource utilization. A methodology for evaluating this cost as it applies to multiply synchronized devices is discussed in the following section.

Device Data Synchronization

An additional factor to be considered in any approach for managing data across a distributed system, is the identification of data to be synchronized across participating nodes. Any algorithm designed should support synchronization of multiple replicas of a distributed database, ensuring consistency of data between all copies of the database. Issues which come to fore include:

Storage constraints on portable devices precludes working with the full dataset on the device; i.e., participating nodes

may not necessarily possess the entire data set of the database, but only an operationally (or geographically) relevant subset.

For disparate data platforms, data will have to be translated or mapped between types. For our research, we have limited ourselves to a consistent database platform across the procured research devices, despite potential differences in underlying hardware topology, in the interests of reducing variability in the evaluation parameters.

[4] Zondervan and Lee, (1999), indicate a preference for using an ID Mapping Table (IMT) to manage data translation between desktop or server databases and mobile device replicas. This was achieved within the context of the above restrictions, by storing the IMT on the main server, and referencing it for translation of data store-relevant documents between device and server. In the multiple-mobile device scenario we envision, using an IMT is less of an issue, as a consistent data platform between the individual devices makes an IMT of limited value (since there is reduced requirement for translation of values between nodes). We can assume therefore that the algorithm is operating as if a 1-1 imaginary IMT mapping exists between all items in a particular device node and any other mobile device against which we want to replicate.

It is necessary to identify the resource costs related to the management of data objects across the distributed nodes of the database. These costs can be segregated into object management costs, communications costs and I/O costs, and calculated using any of a variety of methodologies. For our purposes, the approach outlined by [6] Huang and Wolfson, (1994) will suffice, particularly due to its specificity for determination of object allocation and access costs. In their approach, Huang and Wolfson outlined a methodology for comparison of the competitiveness and costliness (in object resources) of differing distributed object management algorithms (DOMs). Here, competitiveness is a measure of the performance of a particular algorithm, while costliness refers to the resource usage in terms of memory and disk requirements.

In order to comprehensively evaluate the MDD products, this study compared both raw performance characteristics, as well as derived metrics. Criteria for distributed database performance evaluation included I/O, node response-time (database ACK), and real-time node resource usage (memory, disk). The following methodology was used to test and evaluate the candidate set of products:

- Identified performance characteristics data and literature for the candidate products.
- Created a distributed database instance for each product for use with the test infrastructure.
- Implemented a test harness for the MDD product libraries, suitable for evaluating operational and performance characteristics of data transactions against the MDD instances above.

- Performed a set of quantifiable updates and searches against the databases, capturing statistics of performance, as indicated by expressions in previous section.
- Generated cost comparison matrix for the candidate products, using both experimentally derived and literature data above.

4 Product Cost Calculation

We made the following assumptions:

- For any set of read/write operations, the number of nodes that end up with the latest data will be equivalent to the number of nodes in our set. This will be the number of actual devices running a MDD instance for the application.
- It was assumed that all transactions make up the allocation schedule, and the allocation scheme is the set of all devices. It was further assumed that any instance is completely updated after all operations (read & write) have completed. This allowed the remaining assumptions below.
- X = An allocation schedule is an execution schedule where some of the read requests are saving-reads. At the end of the allocation schedule the data object is stored in the local databases of the participating nodes. Then, we can say that X =number of transactions.
- Y = Allocation scheme for a request is the set of nodes (or processors) that have the latest version of the data object in the local database, just before execution. Hence, Y =number of mobile devices.
- Knowing read time, write time from the benchmarks for each product; the average time for any read or write operation against the local device was calculated. Therefore, average read/write time \approx local operation time.
- Finally, we assume that network ping times for benchmark test networks, taken as a fraction of the network communication cost are a good approximation for control message transmission time across the network.

The above assumptions allowed the assignment of values to the specific relations identified:

Fractional control communication ($t_{\text{tcp_bluetooth}}$) = 37.50 ms

$t_{i/o}$ = total time to transmit both data and control (request) message from one device to another.

t_c = time for transmitting control (request) message from one device to another.

t_d = time for transmitting data message from one device to another.

t_{local} = time to store data message on local device.

This gives:

$t_c = t_{\text{tcp_bluetooth}}$

$t_{\text{local}} = \text{average read-write time from benchmark values}$

t_d , = data transmission time

$$= \text{ping transmission time} \cdot \frac{\text{data block size}}{\text{ping block size}}$$

Since, transaction data block size = Integer (8) + String(255)=263 bytes, and ping control message = 32 bytes,

$$t_d = (37.5000 \times 263)/32$$

$$= 308.2 \text{ ms}$$

$$t_{i/o} = t_c + t_d = 37.5 + 308.2$$

$$= 345.7 \text{ ms}$$

Calculating c_c and c_d using the above network communication and benchmark times:

c_c = ratio of the cost of transmitting a control message to the cost of I/O for the object to the local database on secondary storage

c_d = ratio of the cost of transmitting the object between two processors to the I/O cost.

$$c_c = t_c / t_{local}$$

$$c_d = t_d / t_{i/o}$$

5 Data Collection

The candidate J2ME database platforms under consideration in this study were as follows:

- i. Perst
- ii. Berkeley DB Java Edition
- iii. db4o
- iv. J2MEMicroDB

The following table outlines the determination of the product customization factor:

Table 1: Product customization factor determination

	Perst	Berkeley DB Java Edition	db4o	J2ME MicroD B
MDD requirements				
Mobile context	1			1
J2ME support	1			1
Local Autonomy	1	1	1	1
No Reliance on a Central Site				1
Continuous Operation	1	1	1	1
Data Location Independence		1		
Data Fragmentation Independence			1	
Data Replication Independence	1	1	1	
Distributed Query Processing		1	1	

	Perst	Berkeley DB Java Edition	db4o	J2ME MicroD B
MDD requirements				
Distributed Transaction Management			1	1
Hardware Independence	1			1
Operating System Independence	1			1
Network Independence	1			1
Database Independence	1			1
Total Score	9	6	6	10
Customization factor γ^{β}	0.69	0.31	0.46	0.77

The table below outlines the summarized results of the performance evaluation and comparison costs of the candidate products:

Table 2: Derived candidate product comparison costs.

Candidate product	Write time ms	Read time ms	# transactions	Avg. read/write time ms	Customization factor γ^{β}	COST $\beta(\gamma^{\beta})$
Perst	65772	25490	10000	4.56	0.69	49731.13
Berkeley DB Java Edition	0	0			0.31	0
db4o	0	0			0.46	0
J2MEMicroDB	14320	8610	1000	11.47	0.77	2115.65

The first candidate product reviewed, db4o, although a Java-based mobile database, does not support distributed synchronization between mobile instances, without the use of a number of traditional database server components (i.e., installations of a “big-iron” RDBMS such as Oracle or MySQL), as well as the db4o proprietary distributed synchronization manager, dRS.

Similarly, Berkeley DB does not have a mobile Java-based product which provides a distributed capability; as with db4o, this suggests a high customization requirement to port the solution to a mobile J2ME context from the available J2SE framework. This expectation is borne out by the customizability factors determined previously. Additionally, the lack of verifiable public benchmark data for these two products in a mobile context, disqualifies them from further consideration as sufficiently viable mobile distributed database solutions (using the criteria defined for this project).

The other candidate products examined in this part of the project, were Perst and J2MEMicroDB, both of which are

J2ME capable, support multiple node capability and are significantly customizable, due to their open-source licensing regimes. However, a number of items differentiate the two products, particularly from the standpoints of customizability and product maturity. Perst is a well-known, mature product in the distributed database market space, having been first introduced in 2003, and possesses a significant installed base. J2MEMicroDB is a newer product, and does not have the existing uptake that is exhibited by Perst; this may be a result of its academic origins, as it is not heavily promoted as a mobile database solution commercially.

Of further impact, is the large disparity in performance between these two J2ME local databases, without including any distributed capabilities. Perst shows an almost 2-fold order of magnitude speed differential with J2MEMicroD in basic mobile database read/write/update operations (see the tables in the previous section). This is a significant factor, particularly considering the added impact of communication time for database transactions in a distributed context is taken into account. Both Perst and J2MEMicroDB have significant support mechanisms and regular maintenance updates, indicating a vibrant development culture around both. A point of interest is the customization approaches for these products are significantly different, since they have quite distinct approaches in handling database concerns in the limited-resource, distributed environments under consideration. Neither of the products appears to compromise in pursuing highly customizable, developer-friendly usage patterns in the codebases, which bodes well for the MDD development space on a whole.

The conclusion of this evaluation portion of the research, indicates that, of our candidate product set of mobile distributed database solutions, Perst is the most capable MDD solution candidate for distributed information management solutions, as a result of its high level of support, ease-of use, portability, customizability and satisfaction of MDD functional criteria. In particular, Perst, though not a distributed database solution readily capable of multi-nodal input, showed itself to be easily customizable for that purpose, and in fact, was used in the secondary portion of this project, as the base for the MDD prototype application.

6 Prototype Functional Requirements

The prototype application presents a simplified single-page interface, displaying a scrolling list of the most recent posts in the group discussion session which the mobile device is currently monitoring. User/mobile device access is authenticated against a master list for the system; however, there has not been a rigorous application of security protocols in this project. As authenticated devices sign into the system, they will receive the list of current discussion groups, from which one may be chosen to continue communication. Subsequent posts and messages will be maintained within this

group, until the user transfers to another available group (or starts a new one).

It should be reiterated here, that this group discussion system used only participating mobile devices as the backing database store for all operations, i.e., there was no "central" database or external persistent store.

The basic requirements can be itemized as follows:

- User authentication – users should be authenticated against the database for security
- Database node registration – lacking a central database, nodes are registered with each other manually. In a real-world scenario, this would probably be handled using an advertising service component of the application.
- Multi-node data synchronization – data must be replicated/synchronized between all registered mobile devices using the application.
- J2ME (Java Mobile Edition) capable – the application should be packaged and distributed as a portable Java midlet, to illustrate use in multiple device types and environments.

As a prototype, a number of assumptions have been made about the operation and context of the application, highlighted below:

- The application does not attempt to enforce data validation. Invalid input can crash the program, since the application does not enforce real-world restrictions and checks.
- The prototype currently is deployed expecting J2ME HTTP/TCP communication. It has not been designed to communicate over non TCP-based mobile contexts.

Lacking a centralised node tracking database, all nodes have to manually register with each other to support synchronization. In a real-world scenario, a simple solution to this would involve advertising new nodes in a TCP broadcast, or use of a centralized registry.

7 Forum High-Level Design

As concluded in the first part of this project, Perst was identified as suitable for development of high-performance, solution-ready mobile distributed database solutions. This does not suggest that Perst does not have some development limitations. For example, during the development of this prototype, it was identified that the master-slave approach favoured by Perst for supporting replication and distributed database synchronization, would not satisfactorily handle the requirements of the prototype. The master-slave approach enforces read-only capability on all slave nodes, while allowing only the master node to handle updates (writes) to the database. This of course, runs counter to the primary goal of allowing data-entry from any node, with synchronization of all nodes periodically, without need for a centralized server.

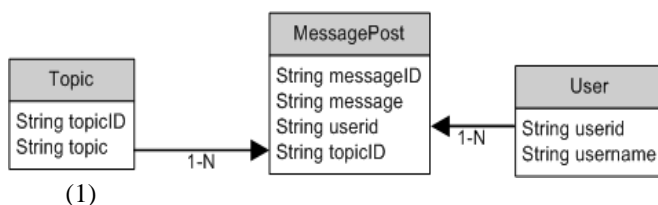
In order to provide our prototype with the ability to support synchronization between the multiple nodes, the application design uses Perst as the underlying local database on each

node, and adds the capability to “register” all participating nodes, with periodic synchronization of updates between all the database nodes. Thus, we have the following sequence of actions for the operation of any instance of the prototype application:

- Local mobile device starts MDDForum application
- Local Perst database is opened
- Register other device nodes with this instance
- Application gets names/addresses of previously registered database nodes
- Application contacts all nodes, and updates itself with most recent data from them
- MDDForum application synchronizes local database with other nodes
- User authenticates against distributed database
- Application presents user authentication screen, and subsequent UI forms.
- Perform regular database operations for forum functionality (e.g., posts, reads, etc.). For purposes of this research, we propose a HTTP-based protocol for communication between database nodes.
- Periodically synchronize local updates with other database nodes.

The prototype application has been designed to utilise TCP/IP for network communication between database nodes, largely because of the ubiquity of support for that protocol in the mobile device space. In a real-world deployment, it is likely that additional flexibility and functionality would be derived from the use of TCP/IP, as this allows the participation of a more diverse set of devices, each of which may operate using differing hardware network interfaces, while participating in the same distributed database. For example, one mobile device may use a 802.11b (Wifi) mechanism for communication, while another participates through the use of a 802.15.1 (Bluetooth) connection. For development and demonstration purposes, this project utilised a wholly TCP/Bluetooth network for node-node communication.

Figure 1: Diagram illustrating entity/table relationships in the prototype..



8 Conclusions

The clearest conclusion to be drawn from the design, development, and performance of this prototype, is that there remains a great deal of work to be done in improving the reliability, flexibility and consistency of distributed databases, especially in the mobile context. Despite the use of a well-known, well-regarded, and highly customizable product as the base for prototype application development (Perst), there were still a myriad of challenges and issues which mitigated against the provision of a robust solution to even the limited scope identified by this two-part project. Our research prototype application performed successfully as designed, with data input and sharing between mobile nodes occurring as expected, within the limits of the parameters set for the application execution, and the defined research examination criteria.

As indicated in the previous section on the prototype design, a number of design decisions were made to reduce the use of certain exception handling and network management operations which might be considered the norm in a non-distributed, immobile environment: the relative immaturity of the underlying technology (J2ME and Perst, turned out to be a significant counterpoint to the achieved aims of code portability and device independence.

Additionally, the requirement to handle the bulk of the distributed database node replication and synchronization logic (as opposed to making use of a database-level capability), significantly impacted the reliability and consistency of the prototype application. As indicated earlier, it appears that for a well-behaved mobile distributed database application, a fairly robust synchronization and node management mechanism has to be developed or delivered with the underlying database.

However, the conclusions of this portion of research on distributed database in mobile devices, are not entirely negative, since it was possible to evaluate multiple products for suitability as infrastructure components in our prototype solution. Further, it was possible to design and develop an actual application that executed on multiple mobile devices and shared data between them, albeit with limited consistency. Finally, this researcher was able to develop custom database node registration and synchronization routines that enhanced the underlying mobile database to support multi-nodal input and replication. It is clear however, that there are a few areas that pose challenges to the significant uptake and usage of mobile distributed database solutions:

- Underlying platform technology and capability must be improved – this speaks directly to the library, connectivity and functionality limitations of environments such as J2ME.
- Distributed database solutions must become much more reliable, in order to provide better capability for managing

real-world scenarios of multi-site data input and data update consistency.

- Customizability of products must be balanced with robust behaviour out of the box – consistency in limited-resource or mobile distributed environments should not require significant development to implement.

It appears likely that we are at the cusp of much more significant development in the mobile distributed database space; personal devices are becoming ever more powerful, with greater connectivity options, making them highly desirable targets for business, commercial and entertainment applications. With this in mind, and based on the limited success achieved with this project's simplified prototype application, it seems highly probable that there will be significant development and improvement in some (if not all) of the products examined in this paper, particularly Perst and J2MEMicroDB. Both of these products are already production-ready, assuming significant custom development; what remains is to reduce the barrier to entry for MDD application developers to make use of these tools in solving future information management problems.

9 References

- [1] Andrew S. Tanenbaum, Marten Van Steen, 2002, Distributed Systems Principles and Paradigms
- [2] Tomasic, A. and Garcia-Molina, H. 1996. Performance issues in distributed shared-nothing information-retrieval systems. *Inf. Process. Manage.* 32, 6 (Nov. 1996), 647-665. DOI=[http://dx.doi.org/10.1016/S0306-4573\(96\)00019-2](http://dx.doi.org/10.1016/S0306-4573(96)00019-2)
- [3] Vijay Kumar (2006), Mobile Database Systems (Wiley Series on Parallel and Distributed Computing) (Hardcover) by Vijay Kumar (2006). ISBN-10: 0471467928
- [4] Quinton Zondervan and Alexandre Lee, 1999, Data Synchronization of Portable Mobile Devices in a Distributed Database System, Quinton Zondervan and Alexandre Lee, Lotus Development Corporation, 1999 ([http://domino.watson.ibm.com/cambridge/research.nsf/0/c71ebac11e6e54f8525661600797829/\\$FILE/mobile.pdf](http://domino.watson.ibm.com/cambridge/research.nsf/0/c71ebac11e6e54f8525661600797829/$FILE/mobile.pdf))
- [5] Motzkin, D. 1991. Distributed database design—optimization vs feasibility. *Inf. Syst.* 15, 6 (Jan. 1991), 615-625. DOI=[http://dx.doi.org/10.1016/0306-4379\(90\)90064-V](http://dx.doi.org/10.1016/0306-4379(90)90064-V)
- [6] Kam-yiu Lam, 2000, Transaction Processing in Mobile Distributed Real-time Database Systems, Kam-yiu Lam, Department of Computer Science, City University of Hong Kong, 2000 (<http://ipdps.cc.gatech.edu/1998/wpdrts/kylam.pdf>)
- [7] Huang, Y. and Wolfson, O. 1994. Object Allocation in Distributed Databases and Mobile Computers. In Proceedings of the Tenth international Conference on Data Engineering (February 14 - 18, 1994). IEEE Computer Society, Washington, DC, 20-29. (<http://citeseer.ist.psu.edu/ACMLINK/http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=655255>)
- [8] Mao, Z. and Douligieris, C. 2004. A distributed database architecture for global roaming in next-generation mobile networks. *IEEE/ACM Trans. Netw.* 12, 1 (Feb. 2004), 146-160. DOI=<http://dx.doi.org/10.1109/TNET.2003.820435>
- [9] Michael Cymerman, 2001, Device programming with MIDP, Part 1 The concepts behind MIDP APIs and J2ME. By Michael Cymerman, JavaWorld.com, 01/05/01 (<http://www.javaworld.com/javaworld/jw-01-2001/jw-0105-midp.html>)
- [10] Qusay H. Mahmoud, 2003, Wireless Application Programming with J2ME and Bluetooth by Qusay H. Mahmoud February 2003 (<http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth1/>)
- [11] Alier, M.; Casado, P.; Casany, M.J., 2007, J2MEMicroDB: a new Open Source lightweight Database Engine for J2ME Mobile Devices by Alier, M.; Casado, P.; Casany, M.J., *Multimedia and Ubiquitous Engineering*, 2007. MUE apos;07. International Conference on Volume , Issue , 26-28 April 2007 Page(s):247 – 252.
- [12] Philipp Bolliger and Marc Langheinrich, Distributed Persistence for Limited Devices, Philipp Bolliger and Marc Langheinrich, Inst. for Pervasive Computing, ETH Zurich, Switzerland
- [13] Hassan Artail, Manal Shihab, Haidar Safa 2008, A distributed mobile database implementation on Pocket PC mobile devices communicating over Bluetooth, Hassan Artail, Manal Shihab, Haidar Safa, Department of Electrical and Computer Engineering, American University of Beirut, P.O. Box 11-0236, Riad El-Solh 1107 2020, Beirut, Lebanon, April 2008
- [14] Eric Falsken , 2008, Enabling the Mobile Enterprise with db4o, By Eric Falsken, db4objects Inc., 2008, <http://www.db4o.com/about/productinformation/whitepapers/db4oWhitepaper-EnablingtheMobileEnterprisewithdb4o.pdf>
- [15] db4objects Inc., 2008, db4o: Java & .NET Object Database - Benchmarks: Performance advantages to store complex object structures, db4objects Inc., 2008, <http://www.db4o.com/about/productinformation/benchmarks/>
- [16] McObject Benchmarks Embedded Databases on Android Smartphone, <http://www.mcobject.com/march9/2009>
- [17] Database Options for the Mobile Application Developer by Bryan Morgan, Jul 19, 2001, <http://www.informit.com/articles/article.aspx?p=22285&seqNum=4>

SESSION

ULTRA LOW POWER DATA-DRIVEN NETWORKING SYSTEM AND ITS REALIZATION

Chair(s)

Prof. Hiroaki Nishikawa

Intermediate Achievement of Ultra-Low-Power Data-Driven Networking System: ULP-DDNS

Hiroaki Nishikawa¹, Kazuhiro Aoki², Hiroshi Ishii³ and Makoto Iwata⁴

¹Department of Computer Science, Graduate School of Systems and Information Engineering,
University of Tsukuba, Tsukuba, Ibaraki, Japan

²Information Infrastructure Laboratory, Inc., Tsukuba, Ibaraki, Japan

³Department of Communication and Network Engineering,
School of Information and Telecommunication Engineering,

Tokai University, Minato, Tokyo, Japan

⁴School of Information, Kochi University of Technology, Kami, Kochi, Japan

Abstract—*Keeping essential communication under the minimum power is crucial in emergency environment. Power consumption will therefore be one of the most important issues to realize both platform and communication environment. This paper reports current status of a research project named "ultra-low-power data-driven networking system(ULP-DDNS)". ULP-DDNS project is aiming at development of data-driven networking system which can achieve ultra-low-power consumption: 1/300 (down to hopefully 1/1000 for final target) less than the present system. This paper first describes the effect of power consumption reduction schemes in ad hoc networking architecture. This paper then demonstrates data-driven implementation of UDP/IP and its power consumption reduction schemes on ultra-low-power data-driven chip multiprocessor(ULP-DDCMP). Furthermore, the authors propose data-driven load balancing scheme to maintain the networking system in working without over-loaded state. Then, this paper describes an implementation of ULP-DDCMP platform simulator and demonstrates its experimental result. Finally, the authors discuss about applying ULP-DDNS to ubiquitous sensor network as the future works.*

Keywords: ultra-low-power, data-driven principle, networking architecture, self-timed elastic pipeline, chip multiprocessor

1. Introduction

Recently, power saving schemes is widely studied in various field. They are especially payed attention after the tohoku earthquake in Japan. It is also important to secure communication environment in emergency such as the earthquake. So, it is important to reduce power consumption as well as effective network processing because continuous communication is needed in emergency [1].

Currently, so-called pervasive networking environment as social infrastructure has been widely studied [2]. Furthermore, there are many studies about mobile ad hoc network [3] 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 [4].

Then, our research project is aiming at development of data-driven networking system which can achieve ultra-low-power consumption: 1/300 (down to hopefully 1/1000 for final target) less than the present system [5]. It is important to realize offloading scheme which means implementation of ultra-low-power networking system without unnecessary power consumption.

To realize the objectives, this project is motivated by a research scenario that is ultimately utilizing passive data-driven principle from networking architecture to a processor platform. This paper describes intermediate achievement of ultra-low-power data-driven networking system(ULP-DDNS). Firstly the authors show the effect of power consumption reduction schemes in ad hoc networking architecture. Secondly, we explain ultra-low-power method in data-driven implementation of UDP/IP. this paper also shows power consumption reduction schemes of ultra-low-power data-driven chip multiprocessor (ULP-DDCMP). In addition, the authors propose data-driven load balancing scheme to keep the networking system in good working without falling into over-loaded condition. Then, validating power consumption is needed to study power consumption reduction schemes in the networking scheme and chip multi-

processor based on the elastic pipeline. This paper therefore describes an implementation of the power simulator/validator which simulates hand-shake in self-timed elastic pipeline and shows its experimental study result. Finally, the authors discuss about applying ULP-DDNS to ubiquitous sensor ad hoc network as the future works.

2. Power Consumption Reduction Schemes of Ad hoc Networking Architecture

Fig. 1 shows phases of data transfer in ad hoc networking architecture. As shown in Fig. 1, there are following phases in networking architecture layer.

- (1) Discovery [6]
- (2) Public key management [7]
- (3) Data transfer in broadcasting [8]

The authors have proposed discovery method which can find target node with minimum the number of search by using global positioning system (GPS). Relay node which search target node for start node autonomously judges whether it can be relay node by its own position which can be get by GPS. Nodes which can be candidates of relay node reply distance between its own node and target node to start node. Start node ask relay node which is minimum distance to target node to search next relay node. Target node is discovered by repeating its search.

Fig. 2(1) shows power consumption of discovery in simple flooding and same one of proposed discovery method. Proposed discovery method reduces data transfer in comparison with simple flooding. Power consumption is proportional to the amount of data transfer. Therefore, power consumption is 25% less than simple flooding as shown in Fig. 2.

When start node wants to establish secure communication to target node, it is necessary to certificate each other. Key management is essential in certification. The authors have proposed power consumption reduction scheme in public key management. Proposed scheme reduce the amount of data in public key management by using trust relationship list.

Fig. 2(2) compares proposed method with conventional method in power consumption. Power consumption in conventional method doesn't depend on the number of established communication. On the other hand, power consumption in proposed method is proportional to the number of established communication. In Fig. 2(2), we assumed that the number of established communication is 20% of all nodes in network as a typical situation. Then, the number of nodes in network is 100. We evaluated that proposed method is 1%

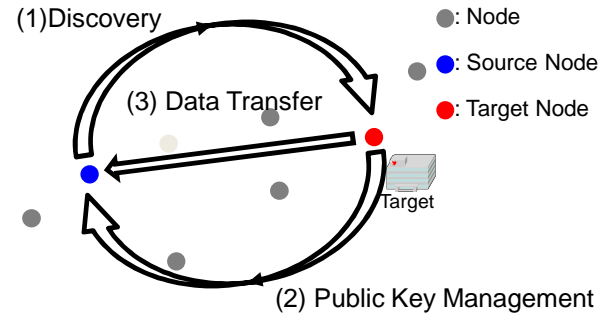


Fig. 1: Phases of networking architecture

power consumption less than conventional method as shown in Fig. 2(2).

Target node sends information to all nodes or certificated node. The authors have also proposed power consumption reduction scheme in flooding. Duplication of relay exists in simple flooding. It causes the increase of power consumption. So proposed method adjust relaying by load of each node (LDCF). Too much load is probably occurred by duplication of relay. Therefore, the node whose load is high stops relaying received information.

Fig. 2(3) compares proposed method with simple flooding in power consumption. Power consumption of proposed method is 25% less than power consumption of simple flooding. In addition, almost nodes in network can get information in proposed method by reducing duplication of relay.

This paper shows estimation of the rate of reducing power consumption in ad hoc mode with these evaluation results. This estimation assumed following situation:

- There is target node near start node relatively. (Discovery method doesn't need to use very much: 0.1% of all communication)
- It is necessary to certificate destination sufficiently. (60% of all communication)
- Flooding is used before certificated communication. (40% of all communication)

The authors estimate that the amount of data is 10% less than conventional method in this situation from each evaluation results. As the future works, we proposed battery-aware counter-based flooding method for long life in communication environment. Refer [8] and [9] about this study.

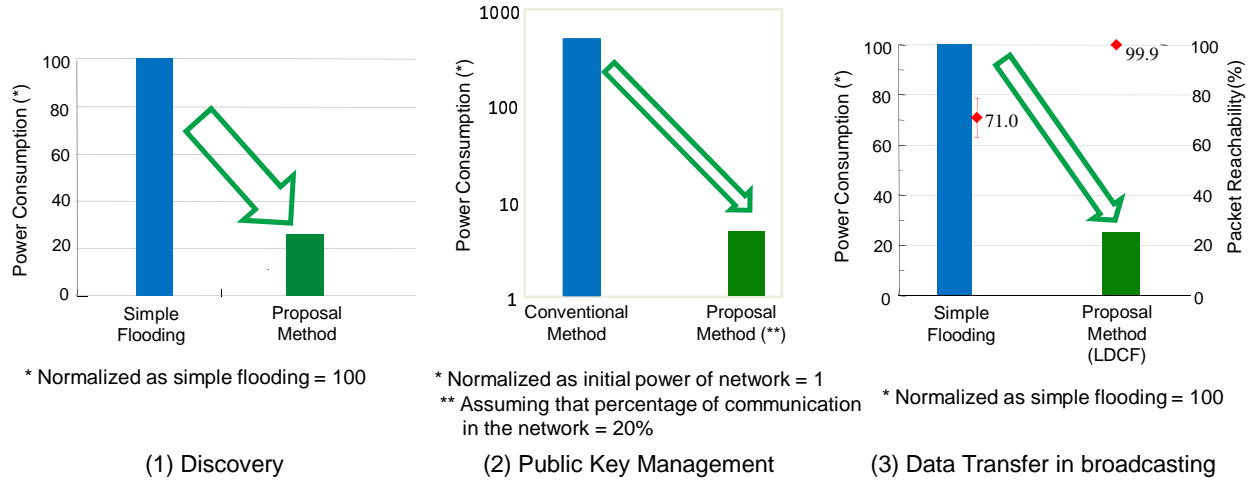


Fig. 2: Evaluation of power consumption reduction schemes on networking architecture

3. Data-Driven Implementation of UDP/IP for ULP-DDCMP

The authors have studied the effectiveness of an implementation of protocol offloader using networking-oriented 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]. Fig. 3 shows data-driven program structure of UDP/IP. We realize header processing and data processing concurrently utilizing multiprocessing capability of data-driven processor without run-time overheads.

This paper shows evaluation of UDP/IP on ULP-DDCMP in power consumption by using power simulator/validator in Section 4.3.

4. Data-Driven Chip Multiprocessor

4.1 Self-Timed Power-Aware Elastic Pipeline:ULP-STP

The authors have studied self-timed elastic pipeline as a VLSI implementation of data-driven principle. Self-timed

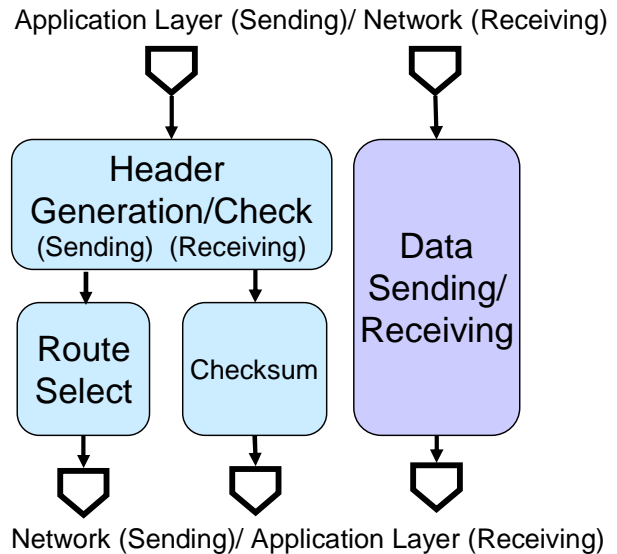


Fig. 3: Data-Driven implementation of UDP/IP.

elastic pipeline has overload tolerance because it is realized by autonomous buffering mechanism. Besides, it has characteristics in ultra-low-power and power control because power consumption of its pipeline is localized in just working parts. Therefore, one of target in this project is implementation of ultra-low-power data-driven chip multiprocessor utilizing self-timed elastic pipeline.

We have studied an implementation of power-gating(PG) function which can cut leak power in waiting by utilizing localization in power consumption of self-timed elastic pipeline[15], [16]. Fig. 4 shows circuits which communicate between pipeline stages in self-timed elastic pipeline. Each

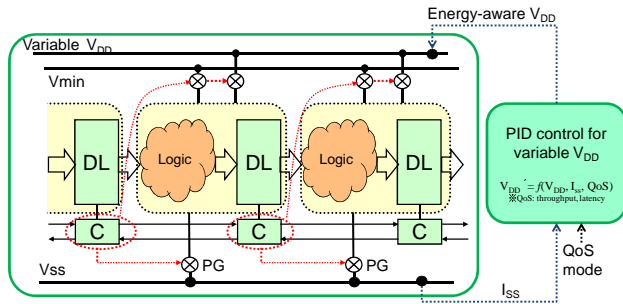


Fig. 4: Power supply control of STP.

stage has PG switches which cut power its logic and data latch to cut leak power in waiting stage.

This paper also shows an implementation of dynamic voltage scaling(DVS) control by using characteristic of observability in power consumption which self-timed elastic pipeline has[17]. Load in self-timed elastic pipeline is proportional to its power consumption. So we can set minimum voltage for demanded performance in suitable interval. Setting voltage derived from proportional-integral-derivative(PID) controller as shown in Fig. 4. The authors named its self-timed power-aware elastic pipeline "Ultra-Low-Power Self-Timed elastic Pipeline: ULP-STP".

The authors evaluated the effect of these implementation by test element group(TEG) on which ULP-STP with PG function and DVS function is realized. Fig. 5 is the graph which shows the effect of PG and DVS functions. X axis of the graph is throughput of the TEG, and Y axis is power consumption of the TEG. The graph shows that power consumption decreases in low voltage when demanded performance is satisfied in each voltage. For example, power consumption in 0.8V(PG-on) is 38% less than power consumption in 1.2V(PG-on). Beside, PG function achieved 93% reducing in waiting leak power ($1.23\text{mV} \rightarrow 84\mu\text{V}$).

Minimizing power overhead in PG function is one of crucial issue in the future works. The authors have been studying sharing power isolator with data latch to minimize its overhead. We also have studied optimizing unit of PG. Because PG in each stage may too large in some situation, we have evaluated changing unit of PG to each module in a stage. Refer [18] about these works.

4.2 An Implementation of Ultra-Low-Power Data-Driven Chip Multiprocessor: ULP-DDCMP

The authors have implemented Ultra-Low-Power Data-Driven Chip Multi Processor(ULP-DDCMP) by utilizing

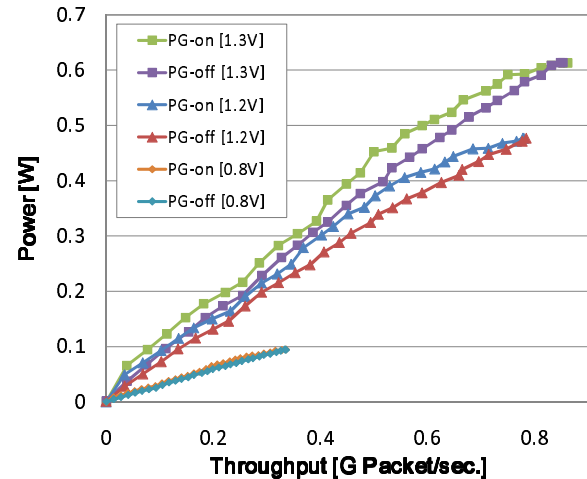


Fig. 5: Relation between throughput and wattage in DVS

ULP-STP technology. Fig. 6 shows configuration of ULP-DDCMP. ULP-DDCMP has 4 ULP-CUE which is data-driven processor core. ULP-CUE is optimized circular elastic pipeline as shown in Fig. 7. It is necessary to change data such as UDP datagram into token which is unit to be processed on ULP-DDCMP. Then off-loading I/F in Fig. 6 is implemented as a interface to ULP-DDCMP. Furthermore, ULP-DDCMP is realized following power consumption reduction:

- Reduction of circuit area
ULP-CUE which is a processor core of ULP-DDCMP is smaller circuit area than CUE-v2 which is conventional data-driven processor [12]. Because matching memory in firing control is implemented smaller than CUE-v2 and circuit for control-driven is removed, the area of ULP-CUE is 50% less than the area of CUE-v2. So power consumption of ULP-CUE is 50% less than that of CUE-v2 because power consumption of processor is proportional to circuit area as shown in Fig. 8.
- Optimized circular pipeline
As shown in Fig. 7, ULP-CUE has circular pipeline for unary operation and circular pipeline for binary operation to reduce power consumption which is not necessary to execute program. Reducing power consumption and execution time is expected because optimized circular pipeline for unary operation doesn't pass firing control stages and shortcuts. It is effective to implemented its loop because unary operation in UDP/IP program is over 80%.
- Chip multiprocessor

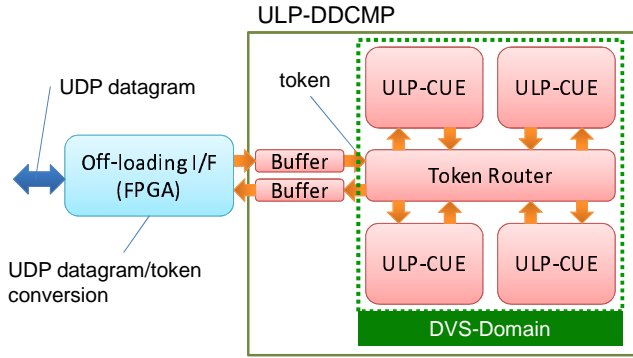


Fig. 6: Data-Driven chip multiprocessor: ULP-DDCMP

Table 1: Spec. of phases in ad hoc networking architecture

Phase	Summary of traffic	Data length of traffic (approximately)
(i) Discovery	Queries to discover nodes and responses	100 byte
(ii) Key management	Public key and public key certification	1k byte
(iii) Data transfer in broadcasting	Media streaming	500 byte

ULP-DDCMP has 4 ULP-CUE in a chip. The authors are aiming at the effect of power consumption in load-distribution by multi-processor. When load in each ULP-CUE is distributed, we think that PG and DVS functions can be utilized. We have estimated demanded concurrency of UDP/IP in ad hoc network in emergency. If data length which is shown in Table 1 is assumed, maximum concurrency is 4. So we implemented ULP-DDCMP which has 4 ULP-CUE because each ULP-CUE is enough to handle single thread.

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. Refer [19] and [20] about load-distributed mechanism and DVS function. In addition, we have developed ULP-DDNS node as shown in Fig. 9. Refer also [19] about this node in detail.

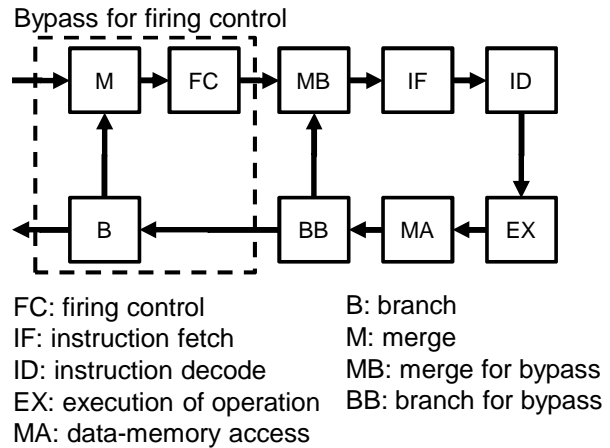


Fig. 7: Data-Driven processor core: ULP-CUE

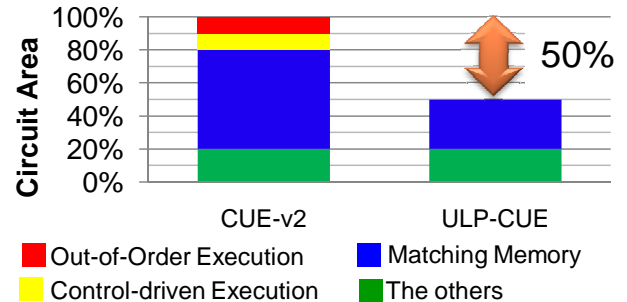


Fig. 8: Comparison between CUE-v2 and ULP-CUE in circuit area

4.3 Power Simulator/Validator for Ultra-Low-Power Data-Driven Chip Networking System

It is necessary to validate power consumption of ULP-DDNS in detail. So the authors proposed power simulator/validator[21]. Then, we have implemented the simulator/validator. This paper shows how to use of the simulator/validator.

Fig. 10 shows pipeline structure in the simulator/validator and validation results. Multi pipeline loop of ULP-CUE is applied to pipeline structure of core in this validation. In first validation(Fig. 10(b)), there is a period of time in which much power is consumed. It is caused by configuration of PID control. Therefore, there is no peak such as first validation in second validation(Fig. 10(c)).

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/validator and network simulator. We think that data log in network simulator can use information of data input in our simulator/validator.

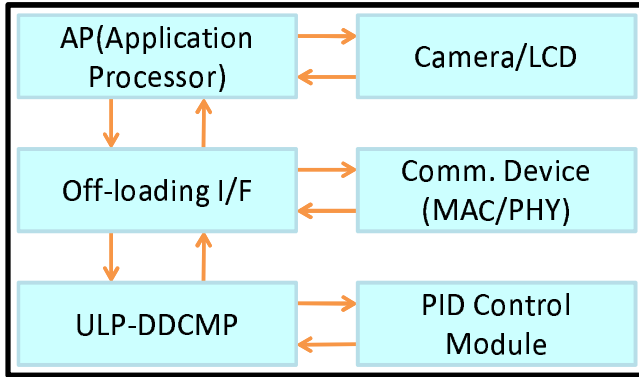


Fig. 9: ULP-DDNS node

On the other hand, specification of CMP in our simulator/validator may apply to a node in network simulator.

Then, it is necessary to adjust parameters in PID control to dynamic voltage scaling mechanism. So the authors have been simulating UDP/IP on ULP-DDCMP architecture in several settings of PID control by using power simulator/validator.

5. Conclusion

This paper described intermediate achievement of ultra-low-power data-driven networking system (ULP-DDNS). We first show the effect of power consumption reduction schemes in ad hoc networking architecture. Their schemes reduce data amount to 1/10 of the present. Because power consumption is proportional to data amount, they reduce power consumption to 1/10 of the present.

Secondly, this paper explained ultra-low-power method in data-driven implementation of UDP/IP. Because turnaround time can be kept minimum by realizing Header handling and data handling concurrently, it is easy to utilize power control scheme on ULP-DDCMP. And this study implemented UDP/IP to apply short cut path to header/data handling.

ULP-DDCMP which is data-driven multi-processor has not only power control scheme but also autonomous balancing mechanism. So we proposed data-driven load balancing scheme to keep the networking system in good working without over-loaded state.

Then, this paper described an implementation of the power simulator/validator which simulates hand-shake in self-timed elastic pipeline and showed its experimental study result. We will study implementing ULP-DDNS utilizing the power simulator/validator.

In the future works, the authors demonstrate the effectiveness of ULP-DDNS using ULP-DDNS node and power

simulator/validator as final result of the project. We also studies about applying ULP-DDNS to ubiquitous sensor network. It is necessary to work in low power and long life in sensor such as observing weather condition. So we will experimental study sensor network on ULP-DDNS utilizing ULP-DDNS nodes. In addition, we also studies about applying ULP-DDCMP to base station. Refer [22] about it.

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] 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.
- [2] Debashis Saha and Amitava Mukherjee, "Pervasive Computing: A Paradigm for the 21st Century," IEEE Computer, Vol. 36, No. 3, pp. 25–31, Mar. 2003.
- [3] Jie Wu, Ivan Stojmenovic, "Ad Hoc Networks," IEEE Computer, Vol. 37, No. 2, pp. 29–31, Feb. 2004.
- [4] 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
- [5] 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.
- [6] 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.
- [7] 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

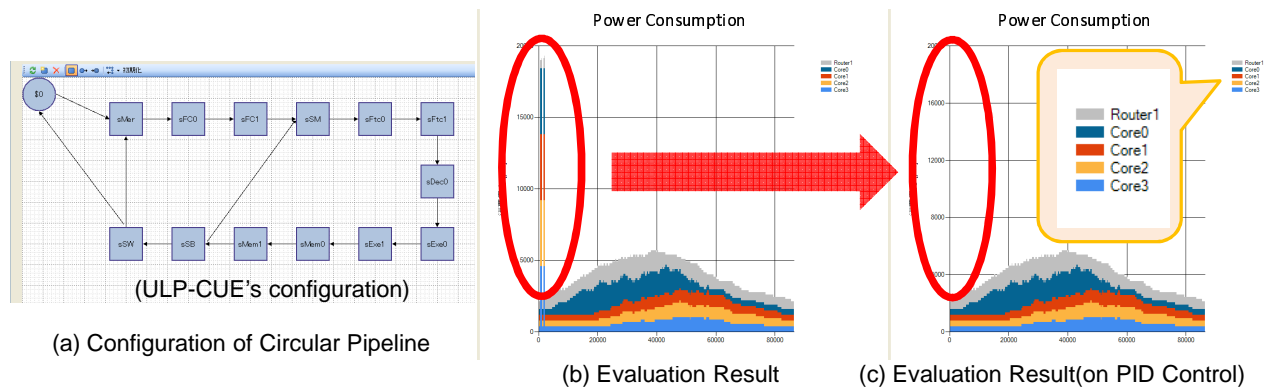


Fig. 10: Power of UDP/IP on ULP-DDCMP

- [8] Keisuke Utsu, Hiroaki Nishikawa, and Hiroshi Ishii, "Broadcast Video Streaming by Load-aware Flooding over Ad Hoc Networks achieving Reduction of Traffic and Power Consumption," Proc. of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP5140, July 2011.
- [9] Keisuke Utsu, Hiroshi Sano, Turganzhan Kassymov, Hiroaki Nishikawa, and Hiroshi Ishii, "Proposal on Battery-aware Counter-based Flooding over Ad Hoc Networks," Proc. of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP5141, July 2011.
- [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] Shuji Sannomiya, Kei Miyagi, Keiichi Sakai, Makoto Iwata and Hiroaki Nishikawa, "Stage-by-Stage Power Gating Circuit for Ultra-Low-Power Self-Timed Pipeline," Proc. of the 2010 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 596–602, July 2010.
- [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] 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.
- [18] Kei Miyagi, Shuji Sannomiya, Makoto Iwata, Hiroaki Nishikawa, "Self-Timed Power-Aware Pipeline Chip and Its Evaluation," Proc. of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP5138, July 2011.
- [19] Shuji Sannomiya, Ryotaro Kuroda, Kazuhiro Aoki, Kei Miyagi, Makoto Iwata, 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, PDP5136, July 2011.
- [20] Yukikuni Nishida, Shuji Sannomiya and Hiroaki Nishikawa, "Multi-Grain Power Control Scheme in Ultra-Low-Power Data-Driven Chip Multiprocessor: ULP-DDCMP," Proc. of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP5137, July 2011.
- [21] Hiroaki Nishikawa, Kazuhiro Aoki, Hiroshi Ishii, and Makoto Iwata, "A Power Simulator/Validator for Ultra-Low-Power Data-Driven Networking System," Proc. of the 2010 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp. 575–581, July 2010.
- [22] Hideki Yamauchi and Hiroaki Nishikawa, "Study on Applying Ultra-Low-Power Data-Driven Processor to Wireless Base Station," Proc. of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP5139, July 2011.

Chip Multiprocessor Platform for Ultra-Low-Power Data-Driven Networking System: ULP-DDNS

Shuji Sannomiya¹, Ryotaro Kuroda², Kazuhiro Aoki³,
Kei Miyagi⁴, Makoto Iwata⁵, and Hiroaki Nishikawa¹

¹Department of Computer Science, Graduate School of Systems and Information Engineering,
University of Tsukuba, Tsukuba, Ibaraki, Japan

²Sharp-engineering Corporation, Osaka, Japan

³Information Infrastructure Laboratory, Inc., Tsukuba, Ibaraki, Japan

⁴Graduate School of Engineering, Kochi University of Technology, Kami, Kochi, Japan

⁵School of Information, Kochi University of Technology, Kami, Kochi, Japan

Abstract—An ultra-low-power networking protocol handling platform is urgently required to realize sustainable ad hoc communication over battery-operated devices in emergent situations without connectivity to the wired network infrastructure. In this paper, the data-driven principle is fully exploited as a basis of ultra-low power because of its on-demand control by which circuits are activated only for processing without any additional controls resulting in power dissipation. Already our previous study reveals that the networking protocol handling contains several sequential processing parts which should be prevented from being the processing bottleneck by efficient instruction executions. This paper proposes an optimized circular pipeline scheme for power-performance-efficient instruction execution especially for the sequential parts, to realize the processor cores of the platform. Based on a prototype LSI implementation of the pipeline structure proposed, the power-performance estimation shows that the reduced amount of power consumption is approximately 33% at most in comparison with the existing circular pipeline structure.

Keywords: data-driven processor, self-timed pipeline, off-load, power gating

1. Introduction

In emergent situations in which network infrastructure is useless, an ad hoc network is necessary to provide communication over mobile battery-operated devices, and lowering power dissipation in such networking devices is urgently required to sustain the communication. With this motivation, the authors of this paper have been studying to realize an ultra-low-power networking platform to which networking protocol handling is transferred or off-loaded, in a conjunction research project, named Ultra-Low-Power Data-Driven Networking System (ULP-DDNS) [1], [2].

To realize the networking protocol handling, processing tasks should be executed simultaneously and independently, i.e., the execution time of each processing task should be

kept minimum to guarantee the quality of communication, even though the input pattern for each processing task is changed depending on the utilization situation of network. To realize such parallel processing in ultra-low-power, a self-timed data-driven chip multiprocessor (DDCMP) is the basis because of its data-driven principle. The data-driven principle realizes parallel processing without any side-effects on the execution time of each processing task because of the absence of the context switching overhead even if the input order and interval are changed. Moreover it also realizes autonomous pipeline stage control which drives logic circuits only in the active pipeline stages. That is, the DDCMP consumes power only for processing but control.

In this paper, to improve the power-performance efficiency of the DDCMP, an instruction execution pipeline is proposed to shorten the processing time of networking protocol handling, and a power gating scheme is also presented to reduce the power dissipation due to the leakage current which is an important issue for VLSI implementation in deep sub-micron era.

Already, in our previous study on the data-driven processor, it is 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 [3]. 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 [4]. 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 pro-

cessor. Therefore, the 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 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. In fact, 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 DDCMP. In this paper, the utilization frequency of the instruction execution modules is focused on, and we propose 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.

2. Requirements for ultra-low-power CMP platform

In our research project, an ad hoc networking node is discussed and designed to realize the ULP-DDNS [5]. The designed node is realized by using the DDCMP and its feasibility will be shown based on a prototype implementation of the node, which is explained in this paper later. Simultaneously, in the project, the power-performance evaluation schemes for the ULP-DDNS are explored from both the theoretical and experimental aspects [6], [7], and they will be implemented into the power simulator whose framework is discussed in [5], to show the quantitative power-performance evaluation. In this paper, the processor core of the DDCMP is focused on and discussed.

In this section, the DDCMP is overviewed, and it is shown that its CMP structure is realized by an instruction execution pipeline in which dynamic power dissipation can be eliminated with no additional circuit. According to the instruction execution pipeline, the requirements to improve the power-performance efficiency are revealed.

2.1 Overview of DDCMP

The DDCMP employs data-driven principle exhaustively from processor architecture to pipeline circuit. As shown in figure 1, the DDCMP is a chip multiprocessor in which data-driven processors are connected with each other via multi-stage interconnection network, and its circuit is fully

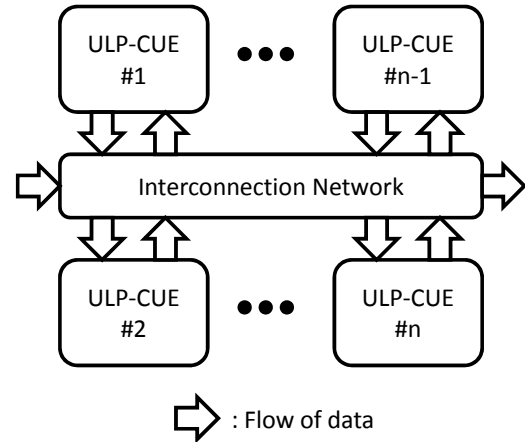


Fig. 1: Data-driven chip multi-processor (DDCMP).

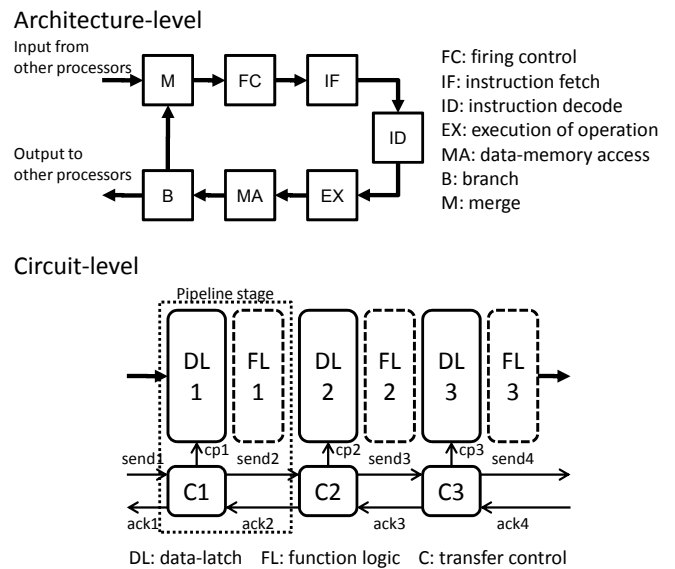


Fig. 2: Instruction execution pipeline of ULP-CUE.

realized by self-timed pipeline. In the DDCMP, the data-driven processor is named ULP-CUE [5].

The ULP-CUE interprets and executes instructions in the order of the arrival of data, i.e., an instruction become ready to be interpreted and executed once its operands arrive. To realize this interpretation and execution, the ULP-CUE is realized by an circular pipeline composed of 5 stages: instruction fetch stage (IF), instruction decode stage (ID), execution stage (EX), data-memory access stage (MA), and firing control stage (FC) which includes a module called matching detection module to detect the arrival of a set of operands. Figure 2 illustrates the block diagram of the ULP-CUE.

In the ULP-CUE, the input data is packed into a token with tag information such as operation code, and the tokens are transferred between the adjacent pipeline stages.

When a token is input, the interpretation and execution of instruction are initiated, and the interpretation and execution are completed after a token laps in the circular pipeline once. The token contains its destination (the address of next instruction) as tag and intermediate processing result in data, i.e., it has processing context in itself, and thus no context switching is required to execute more than one processing tasks in parallel. That is, the processing time of each task in parallel execution is equal to that of the task executed alone as long as the amount of processing load is within the pipeline capacity of the ULP-CUE.

The stages of the ULP-CUE are realized by self-timed pipeline (STP). As shown in the figure 2, each pipeline stage of the STP consists of a data latch for pipeline register, function logic, and transfer control unit named C-element. The data latch, function logic, and C-element are denoted by DL, FL, and C, respectively. The tokens are transferred between the pipeline stages as a result of the communication between the C's in the adjacent stages. The communication is performed according to the 4-phase handshake protocol [8] by using transfer request and acknowledge signals which are called send signal and ack signal respectively. This stage-by-stage transfer control realizes data-driven principle at circuit level, i.e., each pipeline stage is driven in on-demand manner only when a token is transferred. Therefore, signal gating [9] is realized at pipeline stage level naturally.

With the features described above, the ULP-CUE realizes on-demand power consumption comprehensively. That is, the dynamic power is consumed only for processing.

2.2 Utilization of instruction execution stages and intra-stage modules

The data-driven processors realized by self-timed pipeline, such as ULP-CUE, are already implemented for actual applications and their effectiveness is revealed [10]. In those existing processors, the instruction execution stages are deployed over a single circular pipeline, and every instruction is executed on the single circular pipeline regardless of the usage of the instruction execution stage and the intra-stage modules. With this existing circular pipeline, circuit modules commonly used for every instruction execution are overlapped and thus the circuit area can be saved.

However, the usage frequency of the instruction execution stages and intra-stage modules changes depending on the target applications, and some stages and/or modules may be rarely used. In fact, a program description of the UDP/IP protocol handling shows that approximately 88% instructions are unary-operations which can be executed without the firing control (FC) stage and also the number of execution of the multiplication-operation or shift-operation occupies only less than 10% of the total number of instruction execution. This fact reveals the opportunity to improve power-performance efficiency not only by sharing only frequently

used instruction execution stages and but also by powering off the infrequently used intra-stage modules.

Based on the discussions above, it is revealed that the circular pipeline for instruction execution should be sophisticated for networking protocol handling in order to improve power-performance efficiency.

2.3 Power gating scheme

In the pipeline stages without a token in the ULP-CUE, the leakage current flows through circuit. Already, power gating schemes are proposed to reduce the leakage current through such circuit blocks unused for processing.

The power gating scheme realizes the connection and disconnection of the power line to a target circuit by designing the target circuit with MTCMOS (multi-threshold CMOS) structure [11]. Under the MTCMOS structure, the target circuit is composed of fast transistors with low-threshold voltage, meanwhile low-leakage transistors with high-threshold voltage, which are called power switch (PS), are placed between the power line and the target circuit. With the MTCMOS structure, the supplied voltage to the target circuit can be zero by switching off the PS, while the voltage is supplied to the target circuit by switching on the PS. Moreover, in the MTCMOS structure, an isolation cells (ISO) are placed at the output terminals of the target circuit to stop the propagation of the electrically unstable signals from the target circuit powered off.

When the target circuit is powered off, the voltage between the ground-side terminal of the target circuit and the actual ground line is charged by the leakage current through the PS. The voltage between the ground terminal and the ground line is called virtual ground. After the target circuit is powered on, the voltage is discharged and a current called rush current flows, as a result, the voltage between the V_{DD} line and the ground-side terminals of the target circuit becomes equal to the supplied voltage and then the target circuit is ready to process. The time from the power-on to the completion of the voltage supply is called wake-up time.

Already some power gating schemes are proposed, and they are different in the size of the target circuit. The power gating at processor-level [12] can power-off a whole processor core, but the size of the PS should be smaller to suppress the large amount of rush current resulting in the malfunction of the target circuit, and thus the wake-up time becomes larger due to the small PS and the instruction execution time becomes longer. On the other hand, the power gating at logic-gate-level [13] provides the shortest wake-up time because the size of the target circuit is small, but approximately a half of target circuit should be powered-on to control the PS, and thus the reduced amount of leakage current is small. To realize the power-off of a whole target circuit with small wake-up time, in our previous study, the power gating at pipeline-stage-level is already proposed [14]. In our scheme, each pipeline stage can be powered off and

thus the amount of the leakage current through the idle pipeline stages is reduced. That is, the power dissipation in the idle stages unused for instruction execution can be reduced.

Although the power gating at pipeline-stage-level reduces the leakage current through idle stages, the leakage current may flow through busy stages used for processing. For instance, arithmetic-logic units are placed in parallel and some of them are unused for specific instructions, i.e., a shifter module is unused for the execution of an add-operation instruction. That is, the power-performance can be improved by powering off the intra-stage modules according to the usage of the modules.

3. Low-power data-driven instruction execution pipeline

In the previous section, it is discussed that shortening the instruction execution time and reducing the leakage current can be realized by exploiting the usage frequency of the stage and intra-stage modules of the instruction execution pipeline of the DDCMP. To realize this idea, in this section, an optimized circular pipeline is proposed to bypass the stages unused for specific instructions, and then an intra-stage power gating is also presented to power off the infrequently used intra-stage modules.

3.1 Optimized circular pipeline

In the existing data-driven processor, the circular pipeline for instruction execution is structured for binary-operation instructions requiring the FC stage. Unary-operation instructions are considered to be a special case of binary-operation instructions and they are executed on the circular pipeline. However, undesired power and time are consumed and spent when a token with unary-operation pass through the FC stage and thus the power-performance efficiency degrades for the cases where the number of unary-operation instructions occupies most of the programs.

Unfortunately, the biggest circuit in the existing data-driven processors is the firing control stage which occupies approximately a half of the circuit area, and this evidence is shown in this paper later. This is mainly because the matching detection in the firing control stage is realized by using CAM (content-addressable memory) whose circuit library is often unavailable as standard circuit cell library. In the existing data-driven processors, the behavior of the CAM is described at RTL (register transfer level) by using HDL (hardware description language) to make it possible to realize the matching detection only by using standard circuit cell libraries. Consequently, the firing control stage should be activated only when it is required, to improve the power-performance efficiency.

To avoid the activation of the firing control stage, signal gating techniques can be used in the firing control stage to

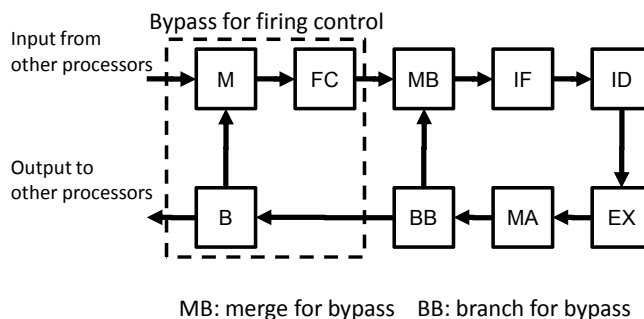


Fig. 3: An optimized circular pipeline structure.

stop the propagation of the signals to the matching detection circuit when unary-operation instruction is executed. However, some additional circuits are required to realize such signal gating, and they not only consumes additional power and but also increase the critical path of the circuit of the firing control stage.

The essence of the data-driven principle is that the instruction execution is started by the arrival of input token. By focusing on this essence, an optimized circular pipeline is proposed to provide different instruction pipelines for the different types of instructions: unary-operation instruction and binary-operation instruction. The optimized circular pipeline makes it possible to overlap only instruction execution stages commonly used for both unary-operation and binary-operation instructions. As shown in figure 3, the optimized pipeline is realized by only adding merge and branch stages to bypass the firing control stage.

With the proposed circular pipeline, the transfer time to pass through the firing control stage can be eliminated from the execution time of the unary-operation instructions. Moreover, the dynamic power dissipation by activating the firing control stage can be zero. Therefore, it can be possible to improve the power-performance of the networking protocol handling whose most part consists of unary-operation instructions.

3.2 Intra-stage power gating

The proposed circular pipeline can shorten the execution time of the unary-operation instructions and reduce the dynamic power dissipation. In addition, a power gating scheme for intra-stage modules is proposed to reduce the leakage current through infrequently used intra-stage modules.

The circuit in the intra-stage modules leaves the sub-threshold leakage current to flow through itself and the power is dissipated. By focusing on the fact that some of them are unused during the execution of an instruction, the power gating is realized at the intra-stage module level by placing the PS between the power line and the intra-stage modules.

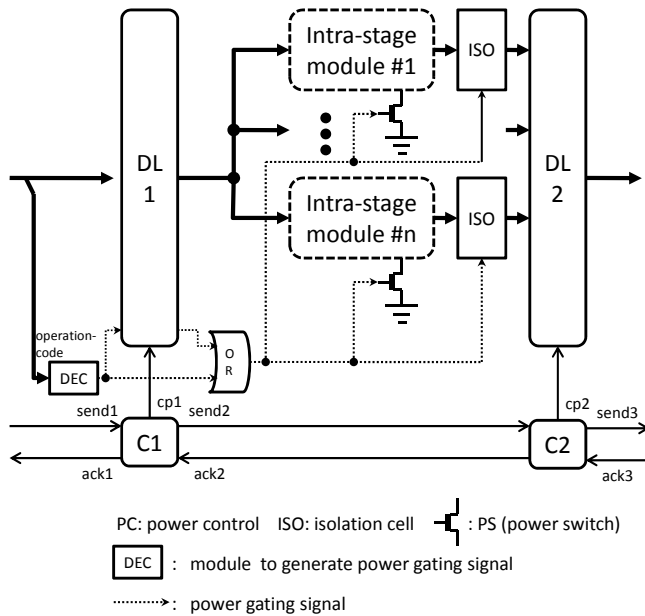


Fig. 4: Intra-stage power gating circuit.

Figure 4 illustrates the circuit structure for the intra-stage power gating. As shown in the figure, the PS and ISO are placed around the target modules and they are controlled by the power gating signal which is generated by inspecting operation-code in the token.

The proposed circuit structure can be realized independently from our stage-by-stage power gating scheme, and thus the leakage current can be reduced more finely, as the result of the synergy between the stage-by-stage power gating and the proposed intra-stage power gating.

4. Evaluation

This section describes the estimation of the effectiveness of the pipeline structure proposed in this paper to improve power-performance efficiency. In our research project, a prototype LSI chip employing the proposed pipeline structure is designed. By using the circuit layout result of the LSI chip, the power-performance ratio of the proposed structure is estimated and compared to that of the existing structure.

4.1 Prototype LSI implementation

In our on-going research project, ULP-DDNS, a prototype LSI chip based on the DDCMP is designed, and the LSI chip is under fabrication process currently. The prototype LSI chip is designed by using 65 nm CMOS process circuit library.

The prototype LSI chip is designed to be able to examine the unique ability of our ad hoc networking node. The unique ability is realized by virtue of the data-driven principle, and it is an ability to avoid the overload situation by monitoring the processing load directly. Although the overload avoidance

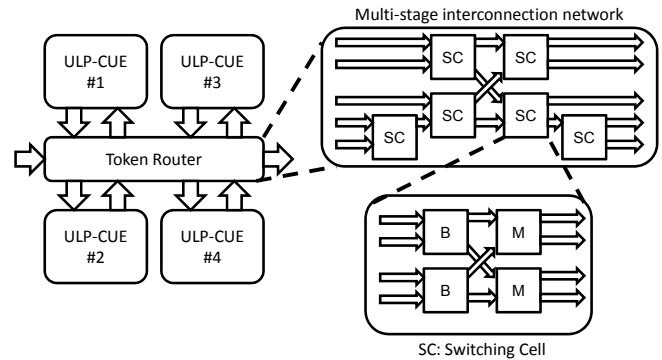


Fig. 5: Prototype of DDCMP.

is important to guarantee the power-performance efficiency because it makes it possible to exploit the processing capability at maximum. However, it is difficult to observe the processing load in currently used non-data-driven processors due to the parallel execution side-effects caused by context switching, and thus a considerably-large headroom margin should be designed in the networking systems based on such non-data-driven processors. Obviously, such margin degrades the power-performance efficiency considerably.

In contrast, the amount of the power consumption of the ULP-STP is in proportion to the amount of processing load by virtue of the on-demand power consumption, i.e., the amount of the processing load in the ULP-STP can be observed only by monitoring the amount of the current flowing on the power line of the ULP-STP outside the DDCMP chip. The framework of the overload avoidance by using this feature and the power-load characteristics are detailed in [5], [7].

To realize the overload avoidance framework, the prototype chip is designed as a DDCMP which consists of 4 homogeneous ULP-CUE cores connected with each other via a multi-stage interconnection network named token router [5]. The designed DDCMP is illustrated in figure 5, and its layout result is shown in figure 6. The die size of the prototype LSI chip is $4.2mm \times 4.2mm$.

The optimized pipeline structure proposed in this paper is used to design the ULP-CUE. The ULP-CUE is designed as a 32-bit data-width self-timed data-driven processor realized by using the proposed pipeline structure. The format of the token of the prototype LSI chip is designed as shown in figure 7. Similarly to the existing data-driven processors [10], the token has the destination node number and generation as tag, in addition to the 32-bit operand data. As for the instruction set, the ULP-CUE provides instructions sufficient to describe the UDP/IP protocol handling which is the target of the off-load in our project. The instructions implemented on the ULP-CUE are typical in comparison with the existing data-driven processors, and they are categorized into 3 types: arithmetic and logical operation type, data-memory access

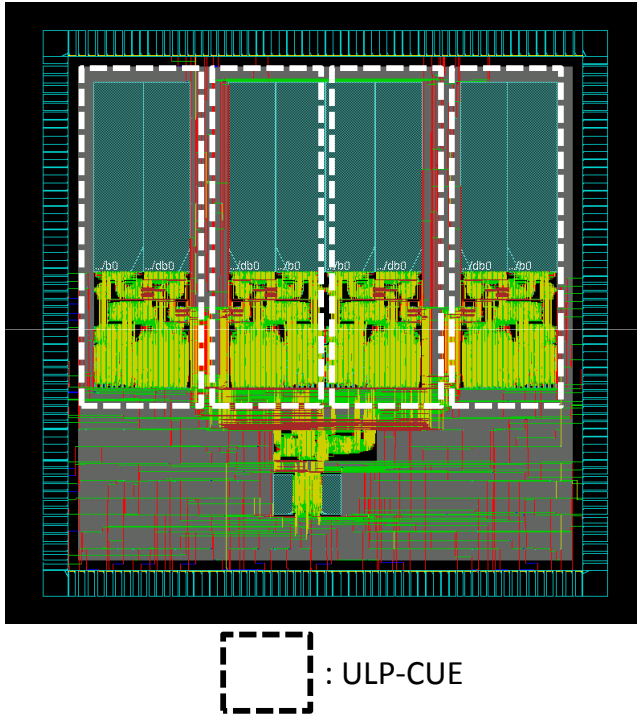


Fig. 6: Layout of prototype LSI chip.

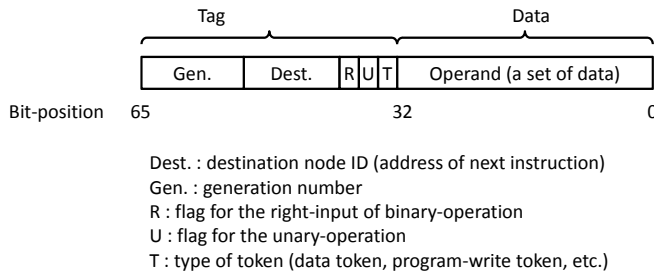


Fig. 7: Format of token in prototype.

type, and generation-manipulation type.

Figure 8 shows the layout of the designed ULP-CUE which has 13 stages, and it also shows that most of the circuit is occupied by the FC and EX stages. The cell area of the FC stage occupies approximately 42% of the total cell area while the EX stage occupies 40%. As for the intra-stage modules, the matching detection circuit occupies approximately 35% of the total cell area while the multiplier and shifter modules occupies 15% and 4% respectively. Fortunately, these large modules are infrequently used in the execution of the UDP/IP protocol handling, and thus those intra-stage modules are selected as the proposed intra-stage power gating.

4.2 Power-performance estimation

As a preliminary evaluation of the proposed pipeline structure, the processing time and the amount of power

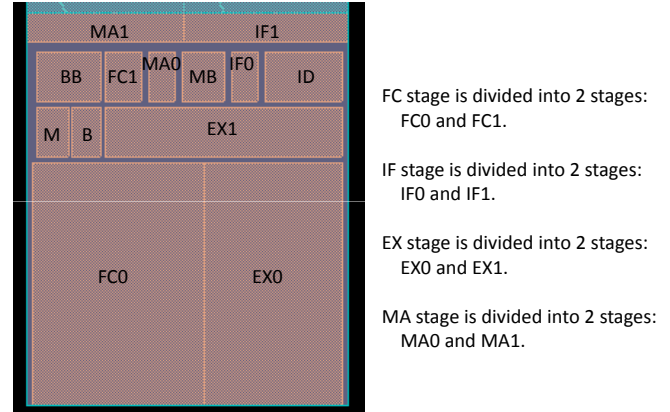


Fig. 8: Layout of ULP-CUE.

consumed are estimated to figure the energy required to execute the UDP/IP protocol handling.

First, the processing time of the UDP/IP protocol handling is measured by using RTL simulation with a placed and routed gate net-list annotated by using the parasitic delay information extracted from the layout result. As a result, approximately 12.8% of the processing time of the UDP/IP protocol handling is reduced by using the proposed pipeline circuits.

The simulation also shows the number of the instruction executions, and the number of the execution of the unary-operation instruction occupies approximately 56.3% of the total number of the instruction executions. Moreover, it is also revealed that the sleep periods of the matching detection module, multiplier module, and shifter module are approximately 14.3%, 99.9%, and 87.2%, respectively.

According to these measured values and the cell area ratio, the consumed power is calculated. In contrast to the existing pipeline structure, the MB and BB stages are added to bypass the FC stage, and the cell area of the pipeline for binary-operation instruction occupies approximately 104% of the total cell area of the existing pipeline structure while that of the pipeline for unary-operation instruction does 56%. Based on this fact, the dynamic power ($P_{dynamic}$) can be roughly estimated as $P_{dynamic} = 104\% \times (1 - R_{uni}) + 56\% \times R_{uni}$ where R_{uni} denotes the instruction execution ratio of the unary-operation instructions. Based on the simulation result described above, the R_{uni} is 56.3% and thus the $P_{dynamic}$ is approximately 77%.

As for the static power consumption, the power consumption by the leakage current increases in proportion to the cell area. The added MB and BB stages occupy approximately 4% of the total cell area, and thus the increased static power ($P_{static+}$) can be roughly estimated as 4%. On the other hand, the leakage current through the matching detection modules, multiplier module, and shifter module can be zero by using the intra-stage power gating. That is, the

decreased static power ($P_{static-}$) can be roughly estimated as $14.3\% \times 40\% + 99.9\% \times 15\% + 87.2\% \times 4\% = 24.4\%$ according to the sleep periods and the cell area described above.

To sum up the estimated values, the ratio of the dynamic power and static power is measured by using an LSI chip fabricated with the same 65nm CMOS process [7], and the dynamic power ratio and static power ratio are approximately 95.6% and 4.4%, respectively. Based on the fact, the power consumption can be estimated as $P_{dynamic} \times 95.6\% + (1 - P_{static-} + P_{static+}) \times 4.4\% = 77.1\%$. By using the process time calculated above, the consumed energy can be estimated as $77.1\% \times (1 - 12.8\%)$, approximately 67%. Consequently, approximately 33% energy is reduced in comparison with the existing pipeline structure.

5. Conclusion

In this paper, a low-power data-driven instruction execution pipeline is proposed to realize the processor core of a chip multi-processor platform for ULP-DDNS. The proposed pipeline exploits the data-driven principle consistently from architecture level to circuit level in order to realize on-demand power consumption resulting in absolutely low power.

To exploit the on-demand power consumption over deep sub-micron VLSI technology with considerable power dissipation due to leakage current through standby circuits, the proposed pipeline provides not only intra-stage power gating reducing the leakage current through idle circuit blocks but also multi-path for shorten the instruction execution time of unary operation instructions issued frequently in networking programs. As a result the power-performance balance is improved for networking processing, and it is estimated that 33% of the amount of power consumption is reduced in comparison with the current instruction execution pipeline.

Already, a prototype LSI chip with the proposed pipeline is designed and its fabrication is now in progress. Meanwhile, we are designing a prototype ad hoc node of ULP-DDNS, and the prototype ad hoc node is realized by not only the prototype LSI chip explained in this paper but also the ultra-low-power schemes proposed in our project in order to show the power-performance efficiency of the ULP-DDNS quantitatively. Based on the measurement on the prototype ad hoc node, we will report the quantitative power-performance efficiency improved by the proposed pipeline structure in another article.

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] H. Nishikawa, H. Ishii, and M. Iwata, "Collaborative research project on ultra-low-power data-driven networking system," Proc. PDPTA'08, pp.697-703, July 2008.
- [2] H. Nishikawa, H. Ishii, M. Iwata, and K. Aoki, "An offloading scheme for ultra low power data-driven networking system," Proc. PDPTA'09, pp.595-601, July 2009.
- [3] S. Ito, S. Nomoto, H. Tomiyasu, and H. Nishikawa, "The Microarchitecture of the CUE-v2 Processor: Enabling the Simultaneous Processing of Dataflow and Control-Flow Threads," Proc. PDPTA'04, pp.525-531, June 2004.
- [4] H. Nishikawa, H. Tomiyasu, M. Okamoto, M. Sugiyama, H. Uchida, O. Mizuno, H. Ishii, M. Iwata, "CUE-v3: Data-Driven Chip Multi-Processor for Ad hoc and Ubiquitous Networking Environment," Proc. PDPTA'07, pp.623-629, June 2007.
- [5] H. Nishikawa, K. Aoki, H. Ishii, and M. Iwata, "Intermediate Achievement of Ultra-Low-Power Data-Driven Networking System: ULP-DDNS," Proc. PDPTA'11, June 2011.(to be published)
- [6] Y. Nishida, S. Sannomiya, and H. Nishikawa, "Multi-Grain Power Control Scheme in Ultra-Low-Power Data-Driven Chip Multiprocessor: ULP-DDCMP," Proc. PDPTA'11, July 2011.(to be published)
- [7] K. Miyagi, S. Sannomiya, M. Iwata, H. Nishikawa, "Self-Timed Power-Aware Pipeline Chip and Its Evaluation," Proc. PDPTA'11, July 2011. (to be published)
- [8] C. J. Myers, "Asynchronous circuit design," Univ. of Utah John Wiley & Sons, Inc., July 2001.
- [9] N. Honarmand and A. Afzali-Kusha, "Low power combinational multiplier using data driven signal gating," Proc. the IEEE Asia Pacific Conference on Circuits and Systems, pp.1456-1459, Singapore, Dec. 2006.
- [10] H. Terada, S. Miyata, and M. Iwata, "DDMP's: self-timed super-pipelined data-driven processors," Proceedings of the IEEE, Vol.87, No.2, pp.282-296, Feb. 1999.
- [11] S. Mutoh, S. Shigematsu, Y. Gotoh, and S. Konaka, "Design method of MTCMOS power switch for low-voltage high-speed LSIs," Proc. Asia and South Pacific Design Automation Conference, Hong Kong, pp.113-116, Jan. 1999.
- [12] T. Hattori et al., "Hierarchical Power Distribution and Power Management Scheme for a Single Chip Mobile Processor," Proc. of Design Automation Conference, pp.292-295, New York, USA, July 2006.
- [13] L. Chen, T. Horiyama, Y. Nakamura, and S. Kimura, "Fine-Grained Power Gating Based on the Controlling Value of Logic Elements," IEICE Trans. Fundamentals, vol.E91-A, No.12, pp.3531-3538, Dec. 2008.
- [14] S. Sannomiya, K. Miyagi, K. Sakai, M. Iwata, and H. Nishikawa, "Self-timed power gating for ultra-low-power pipeline circuit," Proc. PDPTA'09, pp.575-580, July 2009.

Multi-Grain Power Control Scheme in Ultra-Low-Power Data-Driven Chip multiprocessor: ULP-DDCMP

Yukikuni Nishida, Shuji Sannomiya, and Hiroaki Nishikawa

Department of Computer Science, Graduate School of Systems and Information Engineering,
University of Tsukuba, Tsukuba, Ibaraki 305-8573 Japan

Abstract - *The authors are developing multi-grain power control scheme in ultra-low power data-driven chip multiprocessor (ULP-DDCMP) being suitable for a networking process in low power and a high performance. ULP-DDCMP consists of four ultra-low power CUEs (ULP-CUE) and a token router distributing received packets to ULP-CUE by a round robin manner. ULP-CUE consists of an elastic self-timed pipeline (STP) having a flexibility that absorbs a sudden increase of a processing load. Because power consumption of the STP is proportional to processing load, the processing load can easily be assumed by observing power consumption. Moreover, the ULP-CUE has a supply voltage controller to adjust a performance of ULP-CUE to a demanded performance by observing the power consumption. In this paper, we propose a simulation method for developing multi-grain power control scheme in ULP-DDCMP.*

Keywords: *Data-driven chip multiprocessor, Self-timed elastic pipeline, unary operation path, Power control, Energy saving*

1 Introduction

Data-driven processor (DDP) consisting of an elastic pipeline performs a process passively and can perform real-time multiprocessing easily by a buffering feature that the elastic pipeline has in principle [1], [2]. Therefore, DDP is suitable for a process demanded asynchronously like networking process and a process with wide dynamic range of a demanded processing performance.

Moreover, a self-timed pipeline (STP) driven by a handshake control without a clock signal is very low power because pipeline stages processing data only consume power. Additionally, an optimized circular pipeline which has a circular pipeline for unary-operation instruction bypassing a firing control block consuming large power is proposed [3]. By using this STP, unnecessary power is reduced, and a processing turn-around-time becomes short, and power efficiency increases. Especially, the STP is most suitable for a process that unary operation exceeds 80% of whole process as like a UDP/IP packet process.

On the other hand, recently, a manufacturing process miniaturization improves and a sub-threshold leak current consuming power even if a device does not work becomes issue. To solve this issue, power switches are placed at each pipeline stage, a method controlling them by using handshake

signals used for controlling data process has been proposed. The method reduces the sub-threshold leak power drastically because a power is supplied only when a valid data exists in the pipeline stage. However, an overhead becomes issue because it controls the power switches in fine-grain.

As the optimized circular pipeline can localize a data flow in a circular pipeline for unary-operation instruction which is the short cut path, various power control method can be used. For instance, power of pipeline stages in the circular pipeline for unary-operation instruction are controlled concurrently, because data flow in a circular pipeline for binary-operation instruction which is outer pipeline is small, the power of each pipeline stage is controlled per stage,

In order to improve a processing performance, a chip multiprocessor has become mainstream processors currently. However, an observation of each core's processing load and a resource distribution mechanism are needed to distribute processes to each core. These become processing overheads by adopting the chip multiprocessor architecture. As ULP-CUE has a proportional-integral-derivative (PID) control function adjusting a processing performance to demanded performance by controlling a supply voltage according to a demanded processing performance autonomously, it is thought that the processing distribution can be simplified by using ULP-CUE.

ULP-DDCMP [3] can perform multi-grain power control like a chip level, a core level, a pipeline level, and a pipeline stage level. In this paper, a simulation method of a processing load distribution to study a power control scheme of the chip level (a processing load distribution), the core level (PID control) [1] and a STP power estimation method to study a power control scheme of the pipeline level (the circular pipeline for unary-operation instruction) and the pipeline stage level (the power Gating method) are proposed [4], [5]. The power estimation by an emulator for ULP-DDCMP is high accuracy and high speed. Where, the emulator is a prototype of the ULP-DDNS node explained in [3], [6]. However, it is difficult to obtain the power consumption when an application changed, for instance, application change from a wireless node to a home router. The simulator is intended to cover the condition under which the emulator cannot estimate power consumption by an extrapolation or an interpolation of the information gotten by the emulator.

2 Ultra-low-power data-driven chip multiprocessor : ULP-DDCMP

Generally, a chip multiprocessor achieving a high performance at a low clock frequency architecture is used to avoid increase of power consumption caused by an improvement of a clock frequency. In ULP-DDCMP, as shown in Fig. 1, an architecture consisting of four ULP-CUEs and a token router has been proposed. Each ULP-CUE is connected via the token router. Moreover, low power control scheme is studied, too. Tokens received at a token router are sent to some one ULP-CUE and processed

ULP-CUE has a feature that power consumption is proportional to a processing load because it is composed of STP. Thus, STP can know own processing load by observing an electrical current without addition of special functions. By using this feature, optimal power controlling is performed autonomously by measuring an electrical current for a supply voltage and adjusting a supply voltage according to processing load. In other words, when ULP-CUE might become overload state, the supply voltage is increased to avoid the overload state, and when the processing load is low, the supply voltage is decreased and the power consumption is decreased.

Variations of a current consumption and a supply voltage according to a variation of a processing load have delay characteristics. Moreover, a delay from a load variation of ULP-CUE to a supply of an actual desired voltage is caused by a sampling period of the PID controller. In case of a general clocked processor, it does not accept next services until the supply voltage rises and degradation in a quality of service might be caused. On the other hand, as ULP-CUE has a buffering feature in constitutively, ULP-CUE having the tolerance for a momentary overload condition has also a tolerance for the delay describing above.

In case of a chip multiprocessor architecture, it is important how allocate a processing load to each ULP-CUE. In case of thinking about UDP/IP packet processing, One ULP-CUE have to process tokens of UDP/IP packet in each packet, for instance, a check sum of it. Moreover, the load processing each received packet varies because a UDP/IP packet length varies. Because general clock-driven processor does not flexible, as processing load distribution method, a processing load of received UDP/IP packet is distributed to a processor which a processing load is lowest or the processing load is distributed to other processor when a processor becomes high load state as a load distribution method. However, it is necessary to observe the processing load of each processor and calculate a predictable processing load in using these methods. The observation and the calculation cause increasing of the load in overload state.

On the other hand, ULP-CUE consisting by an elastic pipeline has flexibility for the momentary overload described above, and ULP-CUE resolves the overload condition by

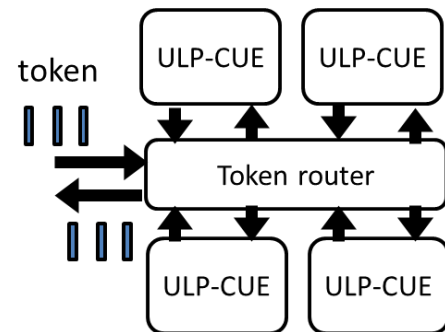


Fig. 1 : ULP-DDCMP

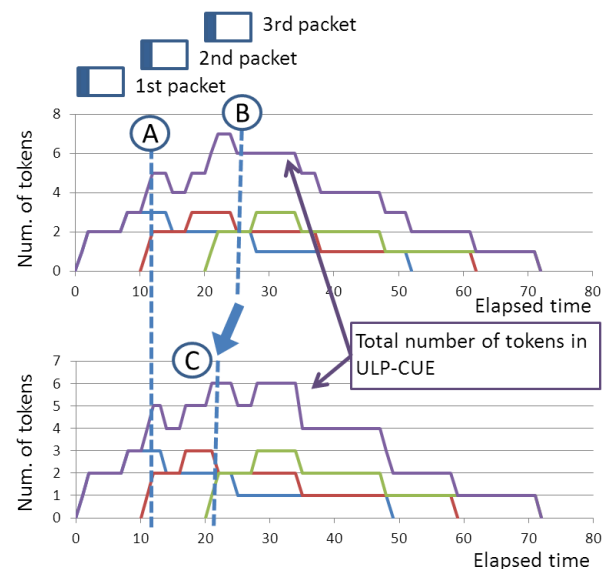


Fig. 2 : Number of tokens at when UDP/IP packets are received.

increasing a processing performance autonomously. Moreover, a round robin manner is proposed as a load distribution method of ULP-DDCMP to avoid an increase of a processing load of the load distributor.

2.1 Load distributor simulation

In previous section, it was described that ULP-CUE varies an own performance corresponding to a processing load condition by using the autonomous power control. Moreover, it was described that a control delay exists in the power controlling mechanism. Received packets are waited to be processed and the packets might be discarded eventually when a control delay exceeds the elastic performance of the elastic pipeline. A delay time not to exceed the elastic performance is clarified and it is necessary to modify a threshold of the voltage control.

UDP/IP packet processes mainly consist of a header processing to decide a forwarding destination obtained by IP header information, a forwarding process to send the IP datagram, and a calculation of header check sum. As the

header processing is heaviest in those processes, the number of tokens in the elastic pipeline varies according to an elapsed time as shown in Fig. 2. When a receiving rate of UDP/IP packets increases and UDP/IP packet gap becomes short, the number of tokens greatly increases, and it becomes seven as shown in Fig.2. Now, it is assumed that ULP-CUE is in an overload state when the number of tokens is seven and a supply voltage is increased when an electrical current value of which the number of tokens is five is observed. In this case, the total number of tokens of first and second packets only have to be less than or equal four before arrival of third packet as shown in Fig.2. Therefore, it only has to shorten a point B to a point C.

However, it takes a time to calculate a delay time of when the supply voltage is controlled dynamically. Therefore, we propose a simulator shown in Fig.3 as it is necessary to estimate a load status and a power consumption of each ULP-CUE. This simulator consists of a UDP/IP packet length generator simulating UDP/IP packet receiving, a packet distributor to distribute the received packet to each ULP-CUE, and core block simulating a behavior of ULP-CUE. The packet length generator has a counter subtracted at an update period of the simulator, and when the counter becomes 0, a fixed packet length, a packet length of pattern generated beforehand, or random packet length is generated, and the packet distributor is called. Then, a simulator time until next packet generation time calculated by the generated packet length is added to the counter. The called packet distributor simulates the round robin behavior proposed in this time, the number of tokens got from ULP-DDCMP platform simulator [6] of each TAT is added to an array expressed the number of tokens in an elastic pipeline of the core block according to the distribution method. Thereby, the number of tokens according to a elapsed time from called time are stored in the token array.

The core block has a current array and a vdd array expressing transitions of the observed current and supply voltage. When the token array is updated and every observing and controlling period of the voltage control block, a func I and a func V are called and they update the current array and the vdd array, respectively. A transition of observed current and a transition of the supply voltage are stored in the current array and the vdd array, respectively.

An update period of the token array is adjusted with a counter similar to the packet length generator, and a TAT or a pipe line tact according to the supply voltage of the vdd array of index 0 is set to the period. Thereby, The load distribution simulation corresponding to Dynamic Voltage Scaling (DVS) becomes possible. Moreover, because the current array and the vdd array are updated every simulation period, to estimate a power consumption by using the current and the voltage of the current array and the vdd array of index 0, respectively.

The structure of load distribution simulator of ULP-DDCMP corresponding to DVS has been proposed [3]. The

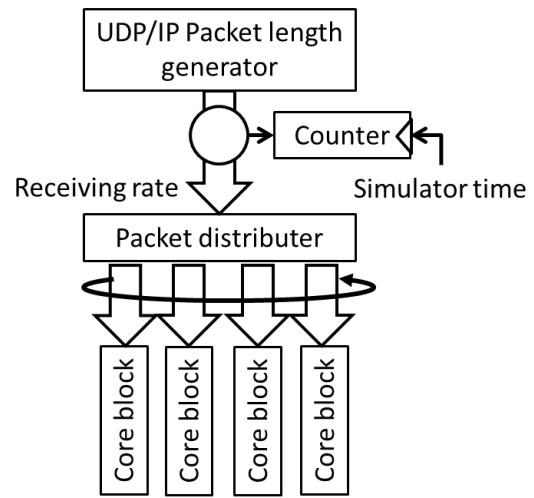


Fig. 3 : Simulation model

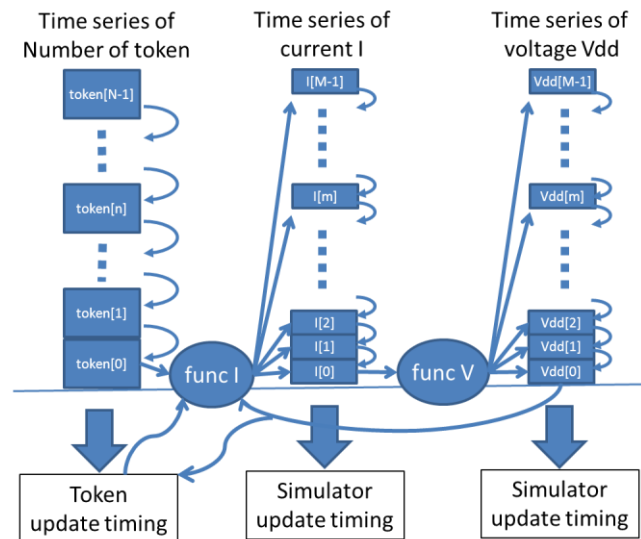


Fig. 4 : Core block simulation model

transition characteristics of the observed current and the controlled voltage is considered in the simulator, The necessary information at the device development such as a upper-limit of control delay time can be gotten. It is possible to study a voltage control algorithm by using an information so that the voltage control is needed how long before an overload condition. Moreover, a variation of a processing load for each ULP-CUE can be observed by observing the number of tonkes in each core

3 Power estimation method

3.1 Optimized circular pipeline

It is necessary to know a power consumption characteristic of ULP-CUE to estimate a power consumption of ULP-DDCMP. An optimized circular pipeline has a

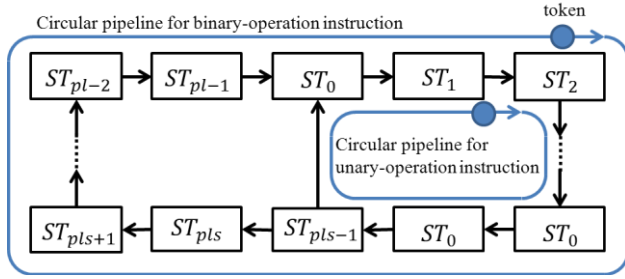


Fig. 5 : An optimized circular pipeline

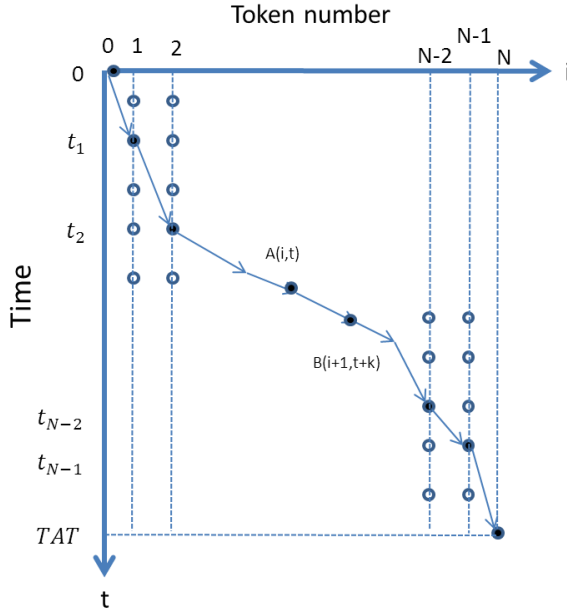


Fig. 6 : Token interval pattern map

circular pipeline for unary-operation instruction to pass a firing control stage of STP with a power gate. Thereby, to estimate the power consumption, it is necessary to consider the circular pipeline for unary-operation instruction of the optimized circular pipeline and a circular pipeline for binary-operation instruction including the firing control stage separately.

As a power control method of the self-timed pipeline, the DVS method minimizing dynamic power consumption while satisfying a demanded performance by varying a supply voltage and the power gating method to reduce a static power consumption caused by a sub-threshold leakage current are mainly proposed. The power gating method turn on or off power switches of each pipeline stage by STP control signals and is implemented with little additional circuit.

The power gating at each stage is possible to reduce the static power consumption significantly because a supply power is turn on or off finely, however, the power gating

consumes a significant switching power needed to switch power lines. Therefore, in this paper, the method that the power gates of the outside stages the circular pipeline for unary-operation instruction of the optimized circular pipeline work at each stage and whole stages of the circular pipeline for unary-operation instruction are switched all together when a token is in the circular pipeline for unary-operation instruction is also examined.

Fig. 5 shows a model to estimate power consumption. The number of stages in an circular pipeline for binary-operation instruction is assumed pl , the number of stages in an circular pipeline for unary-operation instruction is assumed pls . Turn-around-time of the circular pipeline for unary-operation instruction and the circular pipeline for binary-operation instruction are assumed $STAT$ and TAT , respectively. A time necessary so that a token passes each pipeline stage is assumed T_{fi} . Relations of them are expressed in equation (1), (2).

$$TAT = \sum_{i=0}^{pl-1} T_{fi} \quad (1)$$

$$STAT = \sum_{i=0}^{pls-1} T_{fi} \quad (2)$$

3.1.1 Pipeline stage level power gating

At first, a power consumption P is modeled like equation (3) when the power gating of each pipeline stage is performed.

$$P = \frac{sN}{STAT} \sum_{i=0}^{pls-1} A_i V^2 + \frac{(1-s)N}{TAT} \sum_{i=0}^{pl-1} A_i V^2 + \sum_{i=0}^{pls-1} T_{Si} B_i V + \sum_{i=pls}^{pl-1} T_{Li} B_i V + \frac{M_S}{STAT} \sum_{i=0}^{pls-1} C_i V^2 + \frac{M_L}{TAT} \sum_{i=pls}^{pl-1} C_i V^2 \quad (3)$$

where N is the total number of tokens in the optimized circular pipeline, s is a ration of token which is circulating in the circular pipeline for unary-operation instruction in the token in the optimized circular pipeline. T_{Si} and T_{Li} are power supply period to the stages of the circular pipeline for unary-operation instruction and outside of the pipeline, respectively. Similarly, M_S and M_L are numbers of the power switching times in the circular pipeline for unary-operation instruction and outside of the pipeline. Because dynamic power consumption is proportional to a square of a supply voltage,

and static power consumption is proportional to a supply voltage, A_i , B_i , and C_i express coefficients of a dynamic power consumption [7], a static power consumption, a power consumption for the power gating of the pipeline stage i , respectively, finally V expresses the supply voltage.

The supply period of the power to each pipeline stage and the power switching times of each pipeline stage vary depending on the number of tokens and distribution patterns of the tokens [8]. The supplying power to each pipeline stage depending on the power gating is started before the token arrives to certain pipeline stage to avoid performance degradation depending on a delay of a power rising. Thereby, the supply period of the power T_{ON} is expressed by equation (4) when certain token arrives to the pipeline stage i .

$$T_{oni} = \frac{(T_{fi-1} + T_{ri-1})}{2} + (T_{fi} + T_{ri}) \quad (4)$$

However, T_{Si} and T_{Li} become equal to a token period because the power switch is turned ON before the power switch is turned OFF when the token period becomes less than T_{ONi} . The switching times equal to the number of tokens in the pipeline when the token period is longer than T_{oni} , however, the switching times decreases because power is supplied continuously when the token period becomes less than T_{oni}

The distribution of the tokens in STP can have various patterns depending on an application and some kind of influences. For this reason, expected values of the supplying period of a power and the switching times of the power gate are obtained by considering all possible patterns. It is assumed that all distribution patterns are same existence probabilities, and a probability distribution corresponding to the token period is obtained by considering all pattern of tokens placed in STP of when the number of tokens is N . An accumulated probability of exceeding T_{on} described above in the obtained probability distribution is obtained.

When certain token is a head of token moving in the STP, N tokens placed in a TAT is numbered with $0, 1, 2, \dots, N-1$ from the head of token. The tokens can be placed freely in TAT of STP as long as the order does not change, and the number of a combination becomes huge. Thereby, Fig. 6 is referred to obtain the token period. A horizontal axis of the figure represents the token number from the 0-th token which is head of token, and a vertical axis of the figure represents an elapsed time from 0-th token. A sample of the placed pattern of tokens is shown in Fig.6. This sample shows that the first, second, ..., (N-1)-th tokens are placed at t_1, t_2, \dots, t_{N-1} ,

respectively. The N-th token must be placed at TAT because the token moves round in STP during the TAT and the N-th token is necessary to be the 0-th token which is the head of tokens in a next round. The token period is a difference of each elapsed time, for instance, a token period between 0-th token and first token is t_1 , and a token period between first token and second token is t_2-t_1 . Actually, there are many paths like this path, and these paths represent the placed pattern of tokens.

Here, a point $A(i, t)$ and point $B(i + 1, t + K)$ are paid to attention. The point A and B represent the elapsed times of $(i + 1)$ -th token and i -th token, and a token period is K . There are plural path passing between points A and B , and multiplying the number of patterns arriving in token of the point A from a token of point $(0, 0)$ by the number of patterns arriving in a token of point (N, TAT) from the point B gives the number of patterns passing between the point A and the point B as shown Fig. 6. The number $token_f(i, t)$ of patterns arriving in the point A is obtained by equation (5).

$$token_f(i, t) = \sum_{k=0}^{t-T_{min}} token_f(i - 1, k) \quad (5)$$

where T_{min} represents the minimum token period. Specifically, it is the longest handshake period in all pipeline stage, and it expresses $\max(T_{fi} + T_{ri})$. A $token_f(i, t)$ can be obtained by operating this calculation from $i=1$.

On the other hand, the number of patterns arriving the point B from the point (N, TAT) can be obtained by equation (6).

$$token_b(i, t) = \sum_{k=t+T_{min}}^{TAT} token_b(i + 1, k) \quad (6)$$

Similarly, $token_b(i + 1, t + K)$ can be obtained by calculating it from $i = N - 1$ to $i + 1$.

The number of patterns of the token whose period is K is obtained by using these equations. Thus, the number ($inter_t(i, K)$) of patterns of which the period between the i -th token and the $(i - 1)$ -th token is K is expressed by equation (7).

$$\begin{aligned} & inter_t(i, K) \\ &= \sum_{k=0}^{TAT-K} token_f(i, k) \times token_b(i+1, k+K) \end{aligned} \quad (7)$$

In between each token, the number of the token period of K can be obtained by accumulating the number of the token period of K from $i = 0$ to $N - 1$ by equation (8).

$$inter(K) = \sum_{i=0}^{N-1} inter_t(i, K) \quad (8)$$

As described above, the power supply period becomes T_{ONi} when the token period is larger than T_{ONi} . The power supply period becomes the token period when the token period is less than or equal to T_{ONi} . Therefore, the power supply period T_i of a pipeline stage i is obtained by equation (9).

$$T_i = \left\{ \begin{array}{l} \sum_{i=Tmax}^{Toni} K \times inter(K) + \\ Toni \sum_{i=Toni}^{TAT} inter(K) \end{array} \right\} / \sum_{i=Tmax}^{TAT} inter(K) \quad (9)$$

On the other hand, the power supply period T_{Si} is obtained by changing TAT to TAT and calculating a token period because the number of pipeline stages is short. Moreover a power supply period T_{Li} can be obtained by changing the number of token N to sN , similarity.

3.1.2 pipeline level power gating for a circular pipeline for unary-operation instruction

The power gating used to reduce a leak power consumption charge and discharge a power line having a large capacity. Thus, dynamic power consumption becomes large. When a utilization of a circular pipeline for unary-operation instruction is high, it is considered a power consumption is reduced by controlling power gating of the pipeline for unary-operation instruction all together than controlling power gating of each pipeline stage because frequency of charging and discharging the power line are reduced. On the other hand, a power control for each stage is performed because the number of tokens in a circular pipeline for binary-operation instruction than in a circular pipeline for unary-operation instruction is small. Thereby, the static power for the pipeline for unary-operation instruction in equation (10) is always consumed, and T_{LS} becomes one, and the power switching frequency M_S also becomes zero. The power consumption in this power control can be estimated by

equation (10). Other parameters are obtained by the method similar to the stage level power gating.

$$\begin{aligned} P = & \frac{sN}{STAT} \sum_{i=0}^{pls-1} A_i V^2 + \frac{(1-s)N}{TAT} \sum_{i=0}^{pl-1} A_i V^2 \\ & + \sum_{i=0}^{pls-1} B_i V + \sum_{i=pls}^{pl-1} T_{Li} B_i V \\ & + \frac{M_L}{TAT} \sum_{i=pls}^{pl-1} C_i V^2 \end{aligned} \quad (10)$$

Coefficients of these power estimation equations are obtained by using an experimental result of a power consumption and TAT of a test element group (TEG) under certain condition. These estimations are used to obtain power consumption under the other conditions. Then, it is used to interpolate data needed in a simulation using transition characteristics of the power control.

3.1.3 Performance of ULP-CUE

A processing performance of STP is able to express by the number of tokens passing certain pipeline stage, for instance, an execution stage in a unit time. Therefore, the performance Thr is expressed by equation (11).

$$Thr(N, V) = \frac{sN}{STAT} + \frac{(1-s)N}{TAT} \quad (11)$$

STAT and TAT vary according to variations of the supply voltage. Generally, a gate propagation delay for a supply voltage is expressed by equation (12) [9].

$$T_{PD} = \frac{CV}{(V - V_T)^\alpha} \quad (12)$$

where V_T is a threshold voltage of a device, α is a mobility degradation. A processing period T_{fi} of a token in a pipeline stage i is also proportional to $V/(V - V_T)^\alpha$ because logic is composed with gates.

4 Conclusion

The power estimation method to study a multi-grain power control scheme has been proposed. The load distribution simulation method to study power controlling method of the chip level (the load distribution), the core level (PID control), the STP power estimation method to study a power controlling method of the pipeline level (a optimized circular pipeline), the pipeline stage level (power gating) are proposed.

The load distribution simulation method can handle the transition characteristics of the power control corresponding to DVS control, the delay characteristics etc. necessary for designing a chip will be clarified. Moreover, a range in application of very simple round robin method as the load distribution is clarify, and the load distribution simulation method is also used for a development of a power control method to eliminate overhead caused by the load distribution and not to cause overload condition. The STP power estimation method corresponds to the optimized circular pipeline, and it can estimate a power consumption regardless of with or without a circular pipeline for unary-operation instruction. Moreover, the method is able to estimate the power consumption even if only a circular pipeline for unary-operation instruction is controlled at the pipeline level. Therefore, these two method will contribute a study of multi-grain power control method.

In the future, a result of a estimation is compared with a measurement result of actual device, and the estimation method is evaluated for the appropriateness. A clarification of the condition to avoid a overload, for instance, a permissible delay time of power control function and power estimation when a control level of a power gating is changed are performed. We will decrease the power consumption and increase the performance of ULP-DDCMP by using those information

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] 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
- [2] 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.

- [3] Shuji Sannomiya, Ryotaro Kuroda, Kazuhiro Aoki, Kei Miyagi, Makoto Iwata, 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, PDP5136, July 2011.
- [4] Shuji Sannomiya, Kei Miyagi, Keiichi Sakai, Makoto Iwata and Hiroaki Nishikawa, "Stage-by-Stage Power Gating Circuit for Ultra-Low-Power Self-Timed Pipeline," Proc. of the 2010 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, pp.596-602, July 2010.
- [5] 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.
- [6] 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 Technologies and Applications, PDP5135, July 2011.
- [7] Nam Sung Kim, Todd Austin, David Blaauw, Trevor Mudge, Krisztian Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut Kandemir and Vijaykrishnan Narayanan, "Leakage current: Moore's law meets static power," IEEE Computer, Vol. 36, No. 12, pp. 68-75, 2003
- [8] Kei Miyagi, Shuji Sannomiya, Makoto Iwata, Hiroaki Nishikawa, "Self-Timed Power-Aware Pipeline Chip and Its Evaluation," Proc. Of the 2011 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDP5138, July 2011.
- [9] Kevin Nowka, Gary Carpenter, Eric Mac Donald, Hung Ngo, Bishop Brock, Koji Ishii, Tuyet Nguyen and Jeffrey Burns, "A 0.9V to 1.95V Dynamic Voltage-Scalable and Frequency-Scalable 32b PowerPC Processor," Proc. of ISSCC, pp. 340-341, Feb. 2002.

Self-Timed Power-Aware Pipeline Chip and Its Evaluation

Kei MIYAGI¹, Shuji SANNOMIYA², Makoto IWATA¹, and Hiroaki NISHIKAWA²

¹School of Information, Kochi University of Technology, Kochi, Japan

²Department of Computer Science, Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki, Japan

Abstract—*This paper describes an experimental chip of self-timed (clockless) power-aware pipeline incorporating stage-by-stage power gating scheme. Its power gating circuit cuts the voltage-supply to the idle pipeline stages in order to reduce the static (leakage) power dissipation. To reduce the dynamic power dissipation, self-timed pipeline (STP) is one of the suitable circuit architectures because its on-demand transfer control activates only the pipeline stages operating valid data and thus unexpected signal propagation resulting in transistor switching is gated at pipeline stage level. Moreover, each pipeline stage can be slowed down in parallel with processing by introducing dynamic voltage scaling techniques and thus both the dynamic and static power dissipations can be reduced. In this paper, power-performance characteristics of the self-timed power-aware pipeline (ULP-STP) are experimentally analyzed through measuring the actual 65nm CMOS LSI chips and simulating the optimized ULP-STP. The experimental results indicate that autonomous power-awareness of the ULP-STP can save about 48 % power in case of intermittent operation mode.*

Keywords: self-timed pipeline, power gating, dynamic voltage scaling, SPICE

1. Introduction

Low power dissipation techniques of LSI systems are now crucial to realize greener devices, while extracting the full potential of high speed transistors under deep-submicron era. In order to facilitate such low power dissipation 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.

To realize ultra-low-power networking systems, a data-driven chip multiprocessor (DDCMP) architecture has been studied under a collaborative research project [1], [2], [3]. In the proposed architecture, both the dynamic and static power dissipations are minimized by distributing the processing load over multiple processing cores which is slowed down by using dynamic voltage scaling (DVS) technique as long as the required processing speed is satisfied. In addition, the voltage-supply to idle circuit blocks or cores is cut

by using fine-grained power gating (PG) technique. 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) [8].

Because of self-timed elastic data-transfer mechanism of the original STP [10], 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 [7]. This scheme provides natural signal gating [6], 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. In this fabrication, the following design considerations have been made.

- a) Timing constraints on relationship between hand-shake signals and power gating signal,
- b) Minimization of power gating overhead, e.g., isolation elements and power switch transistors, and
- c) Collection of actual parameters that circuit simulation results of any target ULP-STP systems can be compensated more precisely.

As for the remaining part of this paper, the following section describes the ULP-STP circuit and its low-power features. Section 3 discusses the power-performance estimation method of various ULP-STP systems and then introduces an

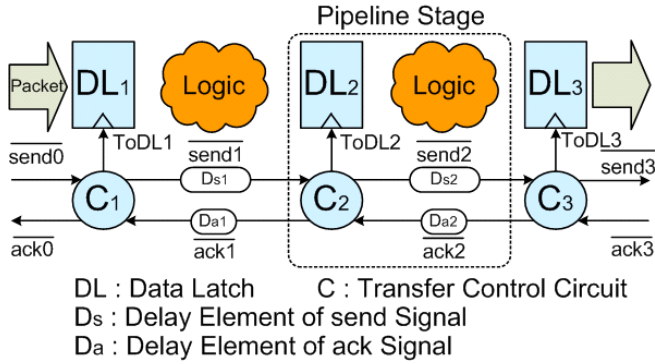


Fig. 1: Self-timed pipeline.

experimental ULP-STP chip indispensable for the proposed estimation method. Section 4 shows the estimation results of an optimized ULP-STP which can be applied to the ULP-DDCMP and then we conclude in the final section.

2. Self-timed power-aware pipeline

The power gating techniques which cut the supply voltage to idle circuits by adding power switch and isolation elements are used to reduce the amount of the leakage current through the transistors in deep sub-micron fabrication technology. Fortunately, the stage-by-stage data transfer controls of the STP exposes which pipeline stages are not processing or idle, and thus the power gating can be realized without any additional mechanisms to manage the all states of pipeline stages. This section describes an ultra-low-power self-timed pipeline (ULP-STP) structure with stage-by-stage power gating and then discusses the power-performance balance of the pipeline structure under the dynamic voltage scaling adaptive to the effective processing load.

2.1 ULP-STP circuit

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 [9] 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.

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.

To realize such stage-by-stage power controls, we have already proposed an ultra-low-power self-timed pipeline (ULP-STP) structure illustrated in figure 2 [7]. 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. Moreover, a power line (V_{min}) providing voltage enough to hold circuit states is routed in addition to the V_{DD} , and two transistors selecting the scaled V_{DD} and V_{min} are added. These added transistors are controlled stage-by-stage by a power control circuit (PC) which observes the send and ack signals. The ULP-STP structure makes it possible to power-down or power-off stage-by-stage, and thus the essential power consumption is concentrated only to the processing stages.

With the proposed structure, the DVS and the PG can be enabled independently or simultaneously. 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

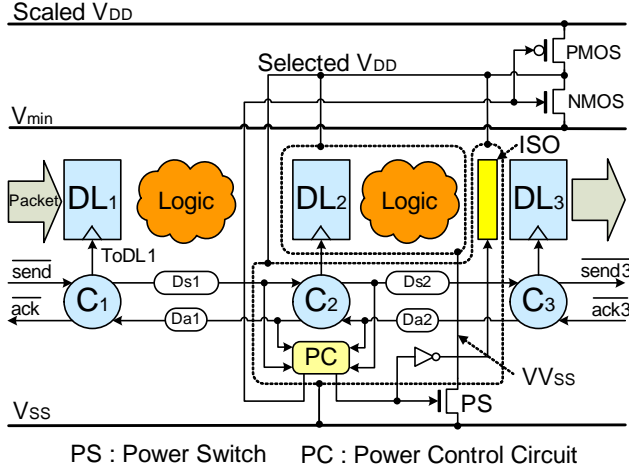


Fig. 2: Power-supply control of STP.

power gating in each stage. Furthermore, the supply voltage to the STP can be autonomously altered without adjusting the clock frequency since the STP itself is clockless. This feature indicates that the STP is more robust than the clocked pipeline, especially in the nanometer-scale processes with more variation in transistor performance.

Although both the DVS and PG of the STP have contribution to reducing the power dissipation, the amount of the leakage current is expected to become larger with the growth of the scaling of transistors. Therefore, in the remaining part of this paper, the stage-by-stage power gating of the STP is focused on.

2.2 Trade-off of stage-by-stage power gating

In the idle stages of the STP, the dynamic power dissipation can be zero because of the CMOS circuit, but the sub-threshold leakage current is carried due to the reduction of the insulating ability of transistors. It is true that the leakage current can be reduced by cutting power line by power switch (PS), but the leakage current is still carried through the PS. In addition, the power is consumed when the PS is turned on and turned off. Also, the isolation elements (ISO's) consumes power when they are driven and they still allows the leakage current through themselves. As for the performance, the resistance of the PS's and the gate delay of the ISO's degrade the processing latency of the STP. Based on these facts, the performance and power is analyzed to take the benefit of the stage-by-stage power gating. Statically, the throughput [*packet/sec.*] depends on the occupancy rate of the STP and it is defined as equation (1), where T_{DL} , T_{Logic} , and T_{ISO} denotes the latency time of the DL, the Logic, and the ISO, respectively, and u means the occupancy rate of a pipeline stage [*packet/stage*]. In comparison with the original STP without power gating, the T_{DL} and T_{Logic} are increased due to the on-resistance component of the PS.

$$Throughput = \frac{1}{T_{DL} + T_{Logic} + T_{ISO}} \times u \quad (1)$$

On the other hand, the total amount of the consumed power ($P_{total}[W]$) depends on the states of the stage, i.e. the occupancy rate u also determines the P_{total} . In a busy stage of the original STP, the power is consumed for switching transistors composing the DL and the Logic, and it is denoted by P_{sw} . In contrast, the power is consumed due to the leakage current through the DL and the Logic in any stage, and it is denoted by P_{lk} . The $P_{total}[W]$ of the original STP is defined as the equation (2), where the pl means the number of the stages of the STP.

$$P_{total} = pl \times (u \times P_{sw} + P_{lk}) \quad (2)$$

In the ULP-STP, two adjacent stages should be powered during the data transfer between the stages to assure the data transferring. To switch the PS, the power is consumed not only for driving the PS itself but also a rush current through the PS. After the PS is turned off, the voltage between the ground terminals of the power-gated circuit and V_{SS} is charged gradually due to the leakage current through the PS. This charged voltage is called virtual V_{SS} (VV_{SS}) and its amount is increased up to V_{DD} . The VV_{SS} is discharged and turned into the rush current through the PS after the PS is turned on. The power consumed because of the rush current is one of the power-overheads of the power-gating. The additional power consumed for switching the PS and the ISO and the rush current of the PS is denoted by P'_{sw} . The PS and the ISO also consumes the power due to the leakage current of them, and it is denoted by P'_{lk} . According to these facts, the amount of the total consumed power of the ULP-STP, P'_{total} is defined as the equation (3).

$$P'_{total} = pl \times \{u \times (P_{sw} + P_{lk} + P'_{sw} + P'_{lk}) + (1 - 2u)(P'_{lk})\} \quad (3)$$

Consequently, power dissipation of the original STP can be reduced by the power-gating but the resistance of the PS must drop the supply voltage to the pipeline stage, i.e., it must lengthen the delay time of the pipeline stage. Furthermore, the ISO increases the pipeline tact of the stage, and PS and ISO consume additional power for themselves. Therefore, it is required to maximize the performance-power efficiency by tuning those additional circuits.

3. Power-performance estimation

As described in the previous section, the stage-by-stage power gating and dynamic voltage scaling techniques introduced to the original STP reduce the power dissipation, but they bring performance and power overheads due to the additional circuits such as the PS and the ISO. Furthermore,

PS in a pipeline stage is kept to be turned on while the stage receives several packets contiguously. This kind of local congestion situation might occur if the number of packets flowing would be relatively close to the upper capacity of the pipeline and if a certain or more pipeline stage(s) would take longer latency time than other stages [11]. Since it is difficult to evaluate those phenomena only by utilizing circuit simulator such as SPICE, we decided to establish a power-performance estimation method that can compensate the circuit simulation results based on actual measurement results of an experimental ULP-STP chip.

In this section, an experimental ULP-STP chip implemented by using 65 nm CMOS process provided by Fujitsu Ltd. is introduced and then a compensation method of circuit simulation results based on the measurement results of the fabricated ULP-STP chip.

3.1 Experimental ULP-STP chip

In order to measure the basic characteristics of the ULP-STP, the specification of the experimental ULP-STP chip is simply defined as long as the performance and power overheads of the additional circuits and the influence of the week congestion situations should be observed under all corner conditions of the LSI, i.e., the best case (-40°C , 1.3 V), the typical case (25°C , 1.2 V), and the worst case (125°C , 1.05 V).

The implemented chip is configured as a ring-shaped circular pipeline composed of 38 homogeneous stages, a merge stage receiving input packets, and a branch stage sending output packets. Each pipeline stage of the chip is composed of a 192-bit DL and 24 8-bit Logic's. The stage accepts and outputs packets with Gray-code data, and each Logic is composed of a Gray-code decoder, adder, and Gray-code encoder. The ISO cell provided by the standard cell library is added to each stage to observe typical overhead of the ISO. As for the PS, the gate length of the NMOS transistor is designed to be 80 nm because its leakage current is minimum against its drain-source current I_{ds} even in the worst case. Figure 3 shows the layout image of the implemented chip composed of a ring-shaped ULP-STP core and a self-testing circuit for its functionality. Furthermore, whether the ULP-STP core operates under the power gating function (PG-on) or not (PG-off) can be selected by asserting or negating its control signal line because it is necessary to compare power-performance of both PG-on and PG-off. The send, ack, and PC output signals at any pipeline stage can be probed from off-chip interfaces.

By using those off-chip interfaces, power-performance characteristics of the implemented ULP-STP core have been measured under various conditions, i.e., $V_{dd} = \{0.8\text{V}, 0.9\text{V}, 1.0\text{V}, 1.1\text{V}, 1.2\text{V}, \text{and } 1.3\text{V}\}$, power-gating mode = {PG-on, PG-off}, operating temperature = $\{25^{\circ}\text{C}, 80^{\circ}\text{C}\}$. Figure 4 shows parts of those measurement results where its horizontal axis denotes pipeline throughput [G packet/sec.(pps)] and

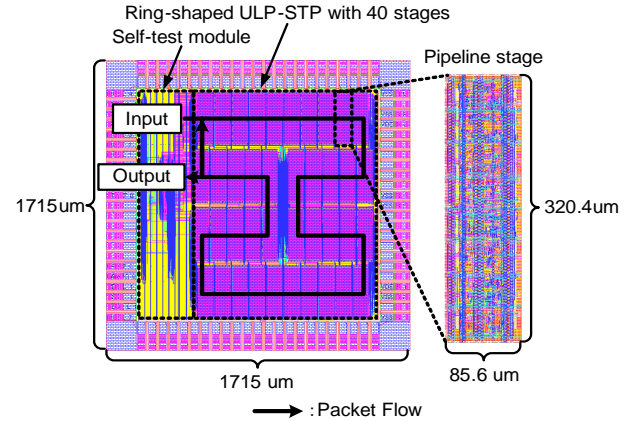


Fig. 3: Layout image of the experimental ULP-STP (65nm CMOS 10ML process).

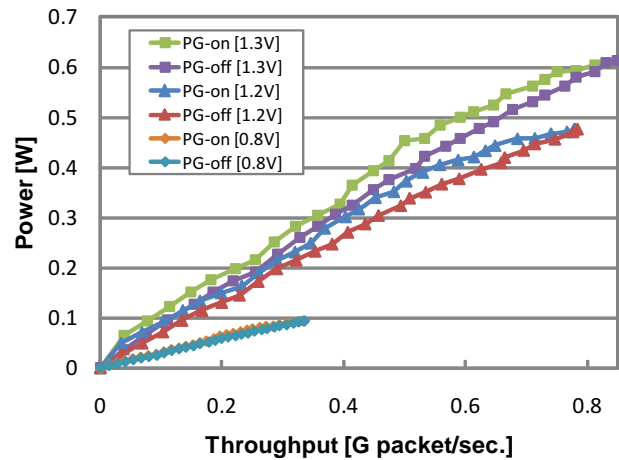


Fig. 4: Performance-power characteristics of the experimental ULP-STP chip under altered supply-voltages (25°C).

its vertical axis denotes consumed power [W]. As shown in the figure, throughput and power overheads due to power gating circuits appears except for fully idle condition, i.e., when throughput is zero. The upper bound of the throughput performance is 0.86 G pps, 0.78 G pps, and 0.33 G pps in case of 1.3V, 1.2V, and 0.8V respectively. The dynamic power is almost proportional to the throughput performance and it indicates that the ULP-STP core can adaptively supply electric power along with its processing load by virtue of its local data-transfer control.

3.2 Compensation of the simulated results

In usual circuit simulation of practical scale LSI, it might take several days at each operation condition. Even for the implemented ULP-STP core, it takes at least two hours only for a single pipeline stage with a single event set of input data. As for whole ULP-STP core, it is therefore

important to estimate its power-performance characteristics based on a small set of the simulation results. Moreover, the simulation result of the ULP-STP core stage slightly involves error in comparison to the actual measurement result of the implemented one. There is a margin of error between them from about 10% to 25%. That error exceeds the typical errors due to the average variations occurred during the chip fabrication process. That means it is necessary to compensate the simulation results in order to evaluate them more precisely.

In order to solve those critical issues, we adopt a simple estimation method of power-performance characteristics for a new ULP-STP system. In this method, the simulation results of a part of a modified ULP-STP system are compensated based on the measurement results of the experimental ULP-STP core as the following:

- 1) To linearly interpolate total switching power and throughput based on a set of simulated power and latency time of several pipeline stages.
- 2) Only in case of the PG-on mode, the ULP-STP might format a few streams of packets which is contiguously flowing in the pipeline. In such cases, the PS and the ISO are kept to be turned on and thus consume no switching power related to their enable signals. The switching power of the PS and the ISO, P'_{sw} , is added to the above interpolated switching power.
- 3) To minimize errors between the interpolated data and the measured data by using the linear least-squares method, in terms of switching power and throughput respectively.

By applying this estimation method to the experimental ULP-STP core, its power-performance characteristics at 25°C can be derived as shown in figure 5. To compare with the actual measurement values, this figure also shows them and indicates the potentiality of the proposed estimation method. Actually, the estimation errors of total switching power are at most 0.1 % in average and those of throughput are at most 0.01 % in average. Although the errors in case of $V_{dd}=1.3V$ are approximately ± 5.0 %, it is lower than the typical process variation. Therefore, it can be said that the proposed estimation method is reasonable and persuadable to analyze various ULP-STP-based systems such as the optimized ULP-STP [8] and the ULP-DDCMP chip [12].

4. Estimation of optimized ULP-STP

As defined in equations (1) - (3), the throughput and the power overheads of the ISO and the PS should be eliminated or reduced to achieve better power-performance in the ULP-STP-based systems.

As for reduction of ISO elements, it has been already proposed that the modified DL can play a role of isolation function [8]. The ISO elements are usually two-input gates provided as a part of a series of standard CMOS gates, and

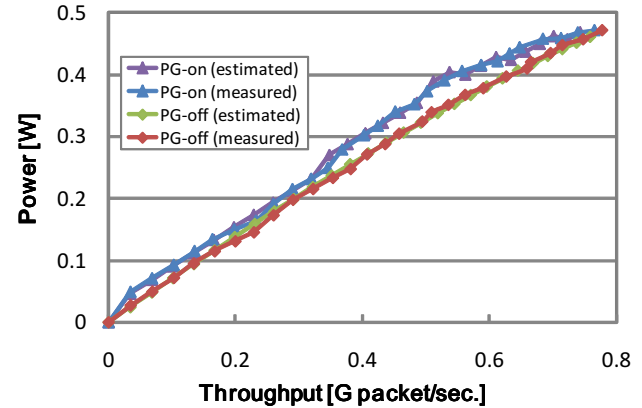


Fig. 5: Performance-power estimation of the experimental ULP-STP chip (1.2V, 25°C).

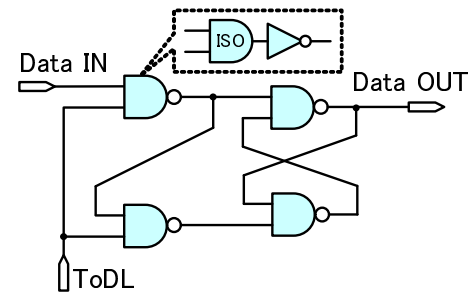


Fig. 6: 1-bit data latch circuit including isolation.

they are categorized into two types: AND-type and OR-type. The former is similar to an AND gate and its data output signal is kept to '0' when its enable input signal is '0', and the latter resembles an OR gate and it outputs '1' when its enable input signal is '1'.

The logical function of the AND-type isolation gate is the same as that of the AND gate. Moreover, the isolating is required only when the DL is powered-on. These facts lead to the inclusion of the isolation gates into the D-latch circuit. Based on this idea, a DL circuit including isolation gate is proposed and it is illustrated in figure 6. A NAND gate of the original D-latch is replaced with one AND-type isolation gate and NOT gate, and thus the ISO can be removed. Although this circuit reduction maintain the signal delay through ISO and DL, the buffer cells for control signal distribution to the ISO can be removed, moreover, the leakage current through isolation gates can be cut because they are also a part of the target circuit of the power gating.

As for optimization of the power switch (PS), the reduction of both the PS itself and the local clock tree to drive the distributed small PS's is important. In this paper, the size of the PS, NMOS transistor, is focused on for the PS optimization, i.e., the gate length of the NMOS transistor is discussed. The longer gate length indicates the larger resistance and capacitance of the PS so that it would

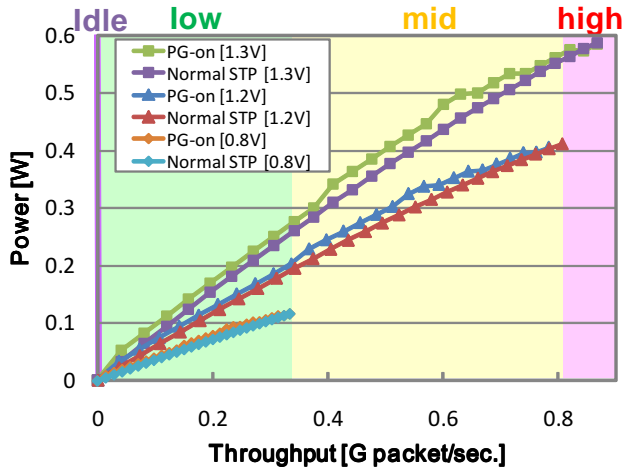


Fig. 7: Power-performance estimation of the optimized ULP-STP (25°C).

lengthen its wake-up time, lengthen the latency time of the pipeline stage, and increase the additional switching power P'_{sw} , while it would decrease the additional leakage power P'_{lk} . If its length is shortened to 60 nm from 80 nm, its capacitance will decrease to 75 %. In contrast, its leakage power will increase, but it can be so small as to be negligible in typical condition (25 °C, 1.2V). Furthermore, since its I_{ds} is proportional to (gate width L /gate length W), I_{ds} increase to 133 %. Therefore, the gate width can be shortened correspondingly so that the total capacitance can be reduced to 56 %.

Figure 7 shows the estimated power-performance characteristics (25 °C) of the optimized ULP-STP which is equipped with both the DL involving the isolation function and the small PS ($L=60$ nm). Because of the isolated DL, the throughput at 1.2V can be improved to 105 %. On the other hand, due to the smaller PS, the degradation of the throughput is at most 2%~3%. As for the power consumption, the breakout of total power consumption is shown in figure 8. In the figure 8(a), the inner circle shows the switching power consumption breakouts of the experimental ULP-STP core, and the outer circle shows those of the optimized one. As show in the figure, the power overhead of the ISO cell can be eliminated and the switching power of the PS can be decreased to 4.07 mW. In consequent, the additional switching power P'_{sw} can be reduced to 40 %. As for the leakage power shown in the figure 8(b), 17 % of the additional leakage power P'_{lk} can be reduced because of the isolated DL and the small PS. However, there are still power overheads due to the local clock buffer tree of the PS (PS_BUF) as shown in the figures and thus the further optimization of the PS_BUF are still remained as a future work.

Since the ULP-STP performs a kind of natural power

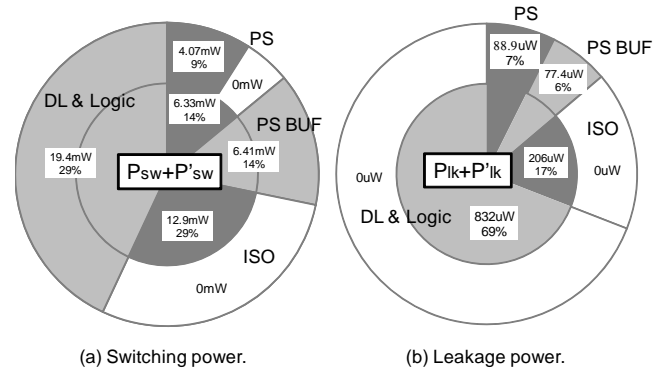


Fig. 8: Power reduction analysis of the optimized ULP-STP(1.2V, 25°C).

supply control along with its processing load at the circuit level, it could play an important power saving role in case of intermittent operations that can be often seen in most of consumer network applications. For example, let us define the processing load of the system as illustrated in the figure 7, i.e., each operation mode of 0.8V, 1.2V, and 1.3V is defined as *low*, *medium*, and *high* traffic mode respectively and also *idle* mode is defined if all pipeline stages at 0.8V have no packet and are gated by the PS. In this case, traffic distribution of the ULP-STP system can be defined as the following equation.

$$idle : low : high = \alpha : (1 - \alpha)(1 - \beta) : (1 - \alpha)\beta \quad (4)$$

Figure 9 shows the energy saving ratio of the optimized ULP-STP against the original STP if α and β are altered. The figure indicates that when the idle time ratio α is greater than 0.6, as smaller β is, as greater the low-energy effect is. This kind situation can be easily imagine if you use your smart phone in daily or when the sensor ad hoc network is operated at some field.

As described in this paper, the advantage of the ultra-low-power STP is that the voltage can be scaled even when the stages processes data and the stage-by-stage power gating can be enabled or disabled independently from the voltage-scaling. That is, the trade-off between the performance and the power can be changed truly dynamically in order to maximize the performance-power efficiency for the target applications' requirements such as a QoS requirement in which the highly prioritized processes should be executed as fast as possible regardless of the amount of power consumption while the processes with low priority can be executed slowly for low-power-dissipation.

5. Conclusion

In this paper, an ultra-low-power self-timed power-aware elastic pipeline (ULP-STP) and its power-performance estimation method are proposed. By fabricating the experimen-

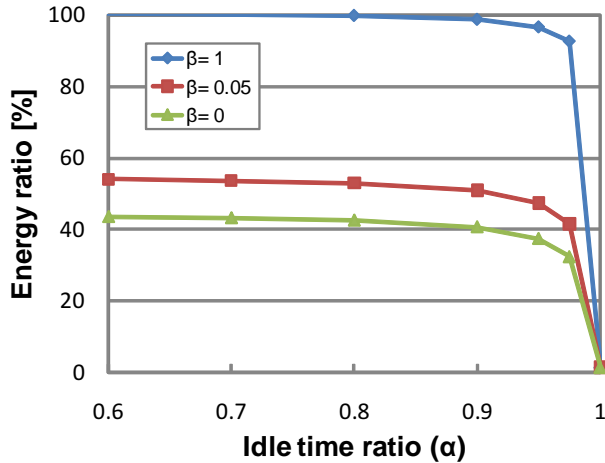


Fig. 9: Relationship between energy reduction ratio and idle time of the optimized ULP-STP (25°C).

tal ULP-STP chip with 65 nm CMOS process, the circuit simulation result can be compensated more precisely based on the actual measurement results of the chip. By utilizing this method, one of our optimized ULP-STP circuits can be evaluated in diverse viewpoints related to its power-performance characteristics. As a result, it is indicated in more quantitative analysis that an ultra-low-power STP equipped with dynamic voltage scaling and power gating can contribute to ultra-low-power LSI platform technologies. That is, it is revealed that the approximately 58% of the power consumption can be reduced with 2% ~ 3% degradation on throughput when the voltage is scaled from 1.3[V] to 0.8[V]. In typical intermittent processing load, the ULP-STP can work with only 48 % energy consumption compared with that of the original STP.

By using the ultra-low-power STP with the proposed circuit, we are now designing a chip multiprocessor (CMP) which is a platform of an ultra-low-power data-driven networking system (ULP-DDNS) discussed in a collaborative research project [1], [3]. The key point of the CMP design is to improve the balance of the performance under its over-loaded region and minimum power for the QoS requests specified network applications by distributing tasks or packets on multiple processing cores equipped with the stage-by-stage power gating and dynamic voltage scaling. To realize such ultra-low-power platform, precisely quantitative analysis tools on power-performance characteristics of individual components are indispensable. In this sense, the proposed estimation method of the ULP-STP could contribute to develop such architecture-level power analysis tools [3].

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] H. Nishikawa, H. Ishii, and M. Iwata, "Collaborative research project on ultra-low-power data-driven networking system," Proc. PDPTA'08, pp.697-703, July 2008.
- [2] H. Nishikawa, H. Ishii, M. Iwata, and K. Aoki, "An offloading scheme for ultra low power data-driven networking system," Proc. PDPTA'09, pp.595-601, July 2009.
- [3] H. Nishikawa, K. Aoki, H. Ishii, and M. Iwata, "Intermediate achievement of ultra-low-power data-driven networking system: ULP-DDNS," Proc. PDPTA'11, PDP5135, July 2011 (to be presented).
- [4] A. P. Chandrakasan, S. Sheng, and R. Brodersen, "Low power CMOS digital design," IEEE Trans. on Solid-state Circuits., vol. 27, No. 4, pp.473-483, Apr. 1992.
- [5] S. Mutoh, S. Shigematsu, Y. Gotoh, and S. Konaka, "Design method of MTCMOS power switch for low-voltage high-speed LSIs," Proc. Asia and South Pacific Design Automation Conference, Hong Kong, pp.113-116, Jan. 1999.
- [6] N. Honarmand and A. Afzali-Kusha, "Low power combinational multiplier using data driven signal gating," Proc. the IEEE Asia Pacific Conference on Circuits and Systems, pp.1456-1459, Singapore, Dec. 2006.
- [7] S. Sannomiya, K. Miyagi, K. Sakai, M. Iwata, and H. Nishikawa, "Self-timed power gating for ultra-low-power pipeline circuit," Proc. PDPTA'09, pp.575-580, July 2009.
- [8] S. Sannomiya, K. Miyagi, M. Iwata, and H. Nishikawa, "Stage-by-stage power gating circuit for ultra-low-power self-timed pipeline," Proc. PDPTA'10, pp.596-602, July 2010.
- [9] C. J. Myers, "Asynchronous circuit design," Univ. of Utah John Wiley & Sons, Inc., July 2001.
- [10] H. Terada, S. Miyata, and M. Iwata, "DDMP's: self-timed super-pipelined data-driven processors," Proceedings of the IEEE, Vol.87, No.2, pp.282-296, Feb. 1999.
- [11] N. Kagawa, S. Sannomiya, and M. Iwata, "Macroscopic power simulation for self-timed pipeline," Proc. PDPTA'10, pp.589-595, July 2010.
- [12] 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. PDPTA'11, PDP5136, July 2011 (to be presented).

Study on Applying Ultra-Low-Power Data-Driven Processor to Wireless Base Station

Hideki YAMAUCHI Hiroaki NISHIKAWA

Department of Computer Science,
Graduate School of Systems and Information Engineering, University of Tsukuba
Tsukuba-shi, Ibaraki, 305-8573 Japan

Abstract – Recent penetration of mobile communication brought high density of mobile phone and need small cell size base station. Power consumption reduction of the base station, especially signal processing unit becomes relatively important. Focusing on the fluctuation of the number of mobile phones in a cell and Ultra-Low-Power Data-Driven Processor (ULPDDP) characteristics which enables linear increase of power consumption with increasing number of process, it is proposed to employ ULPDDP to base station channel unit which processes baseband signal and protocols between mobile phone and base station. Comparing ULPDDP with the DSP which is currently employed in the channel unit, it is estimated that ULPDDP is able to lower 90% of power consumption than DSP.

Keywords: ultra low power, data-driven, base station

1 Introduction

The radio base station in mobile communication has played an important role between radio network and fixed network. It stands between both networks and changes protocols and signals.

To meet the growing penetration of mobile phones and mobile devices, it is necessary to increase accommodation capacity of mobile phones in an area. The maximum accommodation of mobile phones in a cell is limited because of resource constraints limit of radio frequency and bandwidth. To accommodate the increasing number of terminals, it is necessary to scale down the cell size of the base station and increase the number of cells. Meanwhile, mobile network operators are necessary to install a large number of base stations, and in order to facilitate the installation, power saving and downsizing become important subjects as well as cost down for CAPEX and OPEX saving. In addition, a small and low power base station is necessary in order to eliminate the dead zone such as behind buildings and underground.

Regarding the power consumption of compact base station, the wireless transmission of power saving is done by reducing the radio output with reducing the cell size, but the control unit power is independent from cell size. Even though

the cell size shrinks, the same control performance is required and, as the density of cells grows, control unit power issues become relatively important to save total power consumption. In addition, mobile technology improves from 2G, 3G to LTE and more, and the mobile base station is requested to simultaneously support these technologies in order to reduce CAPEX of mobile network operator. The base station requirement becomes severe because of complex signal processing to cover multiple generations of mobile technology, high speed signal processing supporting high speed and short latency communication of LTE, and power consumption reduction.

To satisfy these requirements, developments of semiconductor process technology have been employed to the signal processing. Highly integrated circuits used in signal processing achieve both power saving and complex function. In addition to these efforts, introduction of new architecture have been investigated to achieve further power saving.

The base station is required operating normally under maximum capacity. However, maximum number of mobile phones is not always present there. In general, number of mobile devices in a cell is low enough from maximum capacity and base station has enough room in work load. Maximum capacity is necessary only limited hours. In the base station, required performance depends on the number of active mobile phones. Therefore, it becomes possible to reduce the power consumption of off-peak hours of the number of mobile phone if the power consumption of mobile station changes based on required performance which depends on number of active mobile phone.

In this paper, focusing on the characteristics of ULPDDP which achieves power consumption increases linearly according to processes, it is discussed in which part of the base station ULPDDP is applied.

2 Wireless Base Station

This section describes the current configuration of radio base stations and characteristics of ULPDDP. Considering these description, it is discussed in which part of the wireless

base station is suitable to apply ULPDDP. Also it is estimated how much power saving is realized with ULPDDP.

2.1 Typical Architecture of Wireless Base Station

A mobile phone communication has changed from traditional voice and text to those with photos and videos. Also, the demand which is to access to the Internet content from mobile phone are emerging as a penetration of smart phone. Such change is supported through the rapid improvement of radio access technology from 2G, 3G to HSPA and LTE. Wireless base stations are requested to handle these technologies to support high speed communications. According to the shift of communications from voice to multi media data, mobile telecommunication technology has changed from voice based circuit switching to data base packet switching. In the LTE, voice is transferred by packet with VoIP technology. High throughput packet switching capability is required for the multimedia data. Low latency network is also required to achieve high speed data communication such as TCP/IP. Such data communications need ACK response and the response delay causes low data throughput. The wireless base station is requested high throughput and low latency packet handling capability.

Figure 1 shows a typical configuration of a wireless base station. The primary function of RE is wireless transmission and receive between mobile phone and the wireless signal is converted to base band signal which is transferred between REC via CPRI interface. RE and REC are not necessarily in the same location. For example, they could be several kilometers separated and connected through optical fiber. It is also possible that REC is responsible for several REs. The base band signal sent from REs is processed in facing channel unit. The process includes higher layer processing such as digital signal spreading/despreading, channel coding, frame protocol termination, packet data transfer and voice transfer. [1][2]

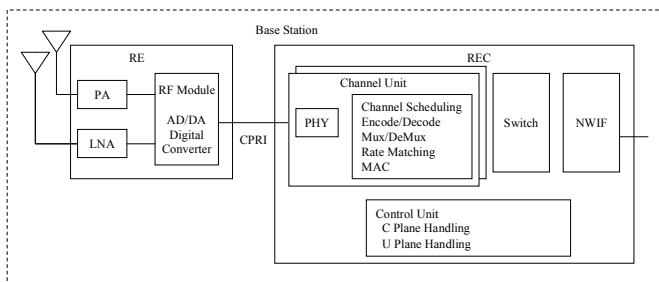


Figure 1. Wireless Base Station Block Diagram

The channel unit and mobile phone send and receive not only user data and voice, but also variety of control information. For example, data rate selection and channel scheduling information is handled by the channel unit to maximize total communication performance under limited

radio wave the resources and under frequently changing the radio communication condition between wireless base station and mobile phone. Also, when a mobile phone enters a new cell, mobile phone registration and handover control information is exchanged between mobile phone and common control unit through the channel unit. Common control unit handles such mobility management and user communication connection management.

The wireless base station is connected to the fixed network through NWIF and communicates with RNC or other network equipment. Switch unit connects channel unit, control unit and NWIF. In order to minimize investment of mobile network operator, the wireless base station at initial stage start minimum frequency channels and increment the frequency channels to meet mobile phone increase. The wireless base station can accommodate multiple channel units to support such flexible capacity change. This also achieves flexible convergence to latest mobile technology.

The channel unit used in the LTE base station currently supports up to 100Mb/s high speed communications and is required high throughput packet processing capability. To optimize radio resource usage with high speed and high quality communications under high speed mobility, channel unit is requested real time channel scheduling and protocol handling. A DSP is usually employed in the channel unit to achieve such realtime and complex control.

A general purpose CPU is commonly used for the control unit. Mobility management, connection management and system maintenance requires less severe response but more complex control than channel unit.

2.2 Wireless Base Station Power Consumption

Figure 2 shows a breakdown of the power consumption of wireless base stations. [3] According to this chart, high power consumption of the wireless base station is the radio power amplifier. In the case, the most important factor to reduce power consumption is increasing the power efficiency of the radio power amplifier.

However, the factor is different when a smaller cell size is introduced. The smaller cell is to increase the number of cell to improve the mobile phone accommodation capacity or to enhance the dead zone coverage with femto cell or pico cell. Because there are resource constraints of radio frequencies, increasing the capacity of mobile phones accommodation per area, it is necessary to lower the radio output power of wireless base stations to reduce the size of the cell. Smaller cell size is able to increase the density of cells in an area. Radio power is reduced by the square of the distance reached. Therefore, the radio power needed to cover the cell and the cell area is proportional. When a cell is divided into individual n pieces of cells, the radio power to cover a piece cell is decreased to 1/n of original radio power. Therefore, the

total radio power needed to cover the area is a constant and independent from cell size. If the radio output efficiency of radio amplifier power consumption remains constant, total power consumption of radio amplifier also remain constant. The smaller the cell size is, the lower power consumption the radio power amplifier is. Power consumption of power supply and air conditioning becomes lower as well.

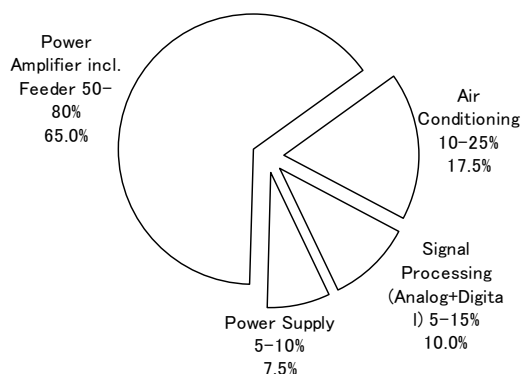


Figure 2. Power Consumption in Wireless Base Station

On the other hand, power consumption of signal processing portion is dependent on the accommodation capacity of the mobile phones and is independent from the cell size. When the cell size is reduced to increase total accommodation capacity, the number of cells increases and the number of signal processing also increases proportionally. Total power consumption of signal processing and control increases. Especially, smaller cell such as femto cell or pico cell which aims to enhance indoor coverage, the power consumption of the signal processing is a non-negligible issue.

In Figure 2, if the cell size of the wireless base station becomes 1/6, power consumption of radio power amplifier also becomes 1/6, however, power consumption of signal processing does not change. As a result, both power consumptions would be approximately the same and power consumption reduction of signal processing would be also an issue. In a recent design of LTE base station, power consumption of power amplifier and signal processing become almost comparable. According to a sample design, power consumption ratio of RE and REC is already 5:3.

In addition to power consumption reduction issue of the small cell size wireless base station described above, there is another problem of power consumption by redundant capacity due to traffic fluctuations. Increasing the capacity of the mobile phone would increase power consumption. Assuming a situation with large fluctuation of number of mobile phone in the nighttime period and daytime, as the business area for example, there is power wasting time zone during which it does not require high processing power. 3GPP is under studying power reduction and focuses on radio power saving

by the scheduled power off and overlay of large cell and small size cell as a practical approach. The power reduction of control system has not yet been studied. [4]

CPUs which are employed in the wireless base station have been developed with a variety of techniques to achieve low power consumption. Shrinking design rule is one of the most popular methods.

Next is to reduce the load on the processor which is caused by the high speed and complex signal processing needs. Specific function hardware engines such as coding and decoding are introduced to reduce processing load. The problem of the method is hardware implementation may consume more energy than software implementation.

The other is, to introduce a hardware mechanism to lower power consumption such as variable voltage and variable system clock. To apply such feature to CPU needs to detect CPU idle state. However, implementing these features are not easy because CPU is requested to maintain response time less than 10mS and to achieve a quick rise of processing power to cope with situations such as a sudden increase in the number of active mobile phones in a cell. It is still seeking effective implementation methods. A simple idle detection and quick response power reduction mechanism applicable to CPU which is employed in channel unit of the wireless base station to support the smaller cells is requested.

2.3 ULPDDP Features

ULPDDP is based on an ultra-low-power data driven chip multiprocessor architecture. An ULP-CUE is an ultra-low-power data driven processor core and the ULPDDP has 4 ULP-CUE. The ULP-CUE is optimized for power reduction to be employed as a network processor and the block diagram of the ULP-CUE is shown in Figure 3. Different from CUE-v2, it has bypass circuit to improve efficiency of unary operation in UDP/IP. [5]

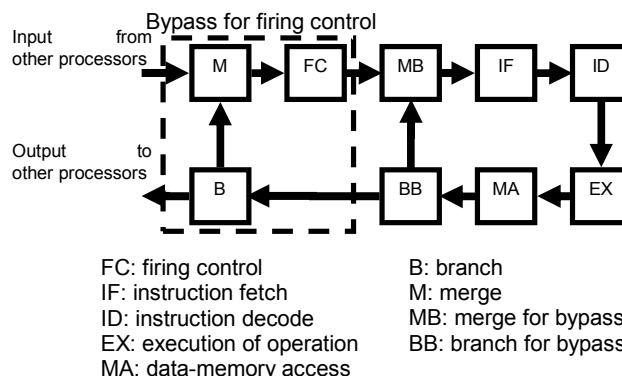


Figure 3. ULP-CUE Block Diagram

In ULP-CUE, each stage is executed asynchronously and connected through Self-Timed-Pipeline (STP) mechanism. Figure 4 shows STP configuration which is employed ULP-CUE. Each STP stage consists of data latch, logic circuit and transfer control circuit (C). Data latch clock is comes from C in each stage and the C is connected to neighbor stage C with asynchronous handshake mechanism. The handshake enables the pipeline progress next stage and also enables the pipeline stop if there is no data to be processed. With this mechanism, data driven and event driven mechanism is realized. The ULP-STP has Power Gating (PG). Idle stage is detected by C status and standby power is cut while the stage is idle. ULP-STP also has dual voltage switching mechanism which enables slower but lower power operation. [6]

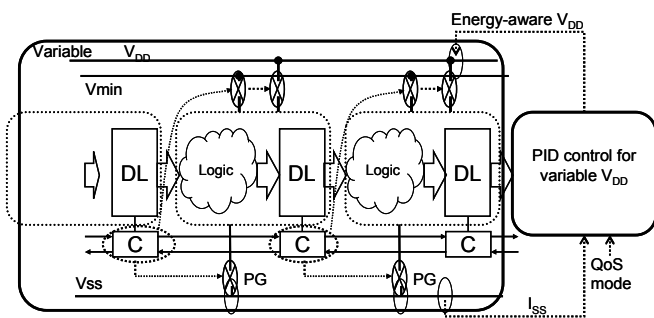


Figure 4. Ultra Low Power Self Timed Pipeline

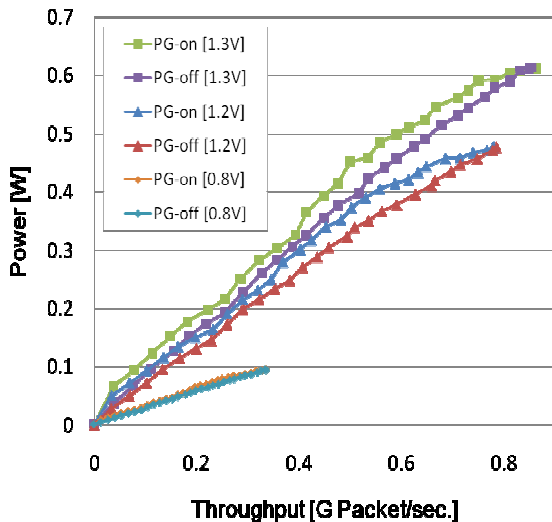


Figure 5 Relation between throughput and power consumption

As shown in Figure 5, the STP mechanism realizes approximately proportional power consumption to the throughput. This means that under low load, ULPDDP

automatically reduces power consumption. This is a desirable characteristic as a processor of wireless base station in a large variation in the number of mobile phone. It is also emphasized that the power saving mechanism is implemented as a data-driven mechanism without complicated control mechanisms. PG and Low-Voltage-Switching effect of power saving is also shown in Figure 5.

2.4 ULPDDP applied wireless base station

As it is described, the wireless base station consists of several components. Although ULPDDP is applicable in the various parts of the wireless base station, it is appropriate to discuss most suitable portion in the wireless base station for early adaptation.

Figure 6 and Figure 7 is a protocol stack between UE (mobile phone) and eNB (wireless base station). [7][8]

In the typical implementation of wireless base station, PHY is realized by hardware in RE and the Channel Unit. MAC and RLC require highly realtime processing and are realized in the Channel Unit. RDCP and RRC require less frequent communication and process than MAC and RLC, so that it is realized by Control Unit. The reason of such demarcation is considering required processing speed of UE communication and easy development of software.

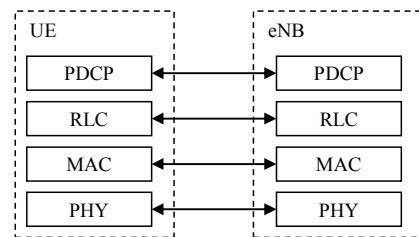


Figure 6. User-plane protocol stack

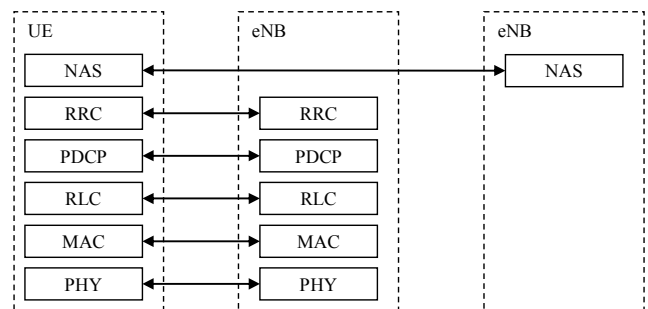


Figure 7. Control-plane protocol stack

Following are also additional implementation the typical implement of each unit in wireless base station.

1. RE's basic function is to send and receive RF signal and conversion to base band. Main power consumption of RE is RF circuit. Sophisticated intelligence of control is also not required because of its simple implementation. Digital

- signal circuit in RE is also requested high speed and hardware logic such as ASIC are deployed.
2. Switch and NWIF are mainly hardware implementation and usually 1 unit is installed in a wireless base station so that its power consumption is independent from the number of mobile phone.
 3. Control Unit also installed 1 unit in a wireless base station and its number is independent from a wireless base station capacity. It has many functions such as wireless base station control, call processing and mobility management. Although some of function processes depend on number of mobile phones, considering complexity of control processor software, merit applying ULPDDP is not much.
 4. Channel Unit handles processes which depend on number of mobile phones such as channel assignment, modulation selection and packet framing and also DSP is a main control of the Channel Unit.

Considering these points, applying ULPDDP to the Channel Unit and replace the DSP is considered as the simplest and the most effective employment of ULPDDP.

2.5 Power Consumption Estimation

Figure 8 shows the comparison of X-Scale which has Von Neumann architecture and CUE-v2 which has similar architecture of ULP-CUE. UDP/IP is used for this comparison and CUE-v2 achieves 61% lower power consumption than X-Scale [9]

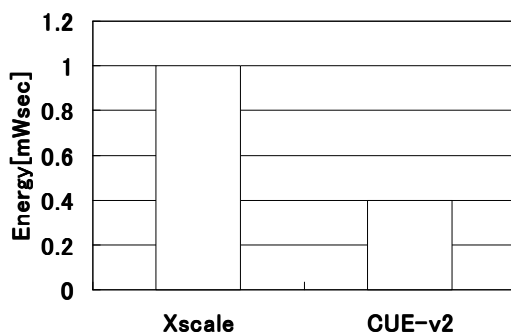


Figure 8 Power Consumption Comparison

UDP/IP is a simple protocol which does not require ACK response for data transfer. On the other hand, Channel Unit communicating with mobile phone needs a response for retransmission control. In the CUE series processor, such protocol is not yet implemented and evaluated. To estimate power consumption, authors set following assumptions.

1. Same energy ratio: Power consumption of X-Scale and DSP is same for the UDP/IP process.
 - Although DSP and X-Scale have different architecture and employ different design rule, the difference is ignored for equal estimation.

2. Same Execution Steps: Both DSP and ULP-CUE need comparable steps to execute. Specifically, same number of ALU in DSP and EX in ULP-CUE are used for both implementation.
 - From operational point of view, same functional operation require
3. Enough concurrent processing: During WAIT state of ACK, there are enough active parallel threads and DSP and ULP-CUE
 - This is regarded as full load comparison.
 - In the partial load environment, ULP-CUE is expected lower power consumption because of ULP-STP.

With these assumptions, same power ratio of Figure 8 is able to be applied to DSP and ULP-CUE comparison.

Power consumption of ULP-CUE is 50% less than CUE-v2. [9] ULP-CUE power consumption is in proportion to throughput, so that if ULP-CUE is not saturated, power consumption of ULP-CUE and ULPDDP which has 4 ULP-CUEs is the same.

Considering these estimation, ULPDDP is expected 10% of power consumption of DSP. In a Channel Unit design for example, DSP consumes around 10W. The Channel Unit with ULPDDP is expected to save 9W which is around 20% of the Channel Unit power consumption.

3 Conclusions

In this paper, it is studied applying ULPDDP to wireless base station to lower power consumption. It is described that ULPDDP has suitable characteristics being employed for the wireless base station, especially channel unit which processes signals and data between mobile phones and wireless base station. In the Channel Unit, DSP has been employed for the signal processing. It is estimated that ULPDDP could lower power consumption 90% than DSP. This estimation is based on assumptions.

A work is needed to find appropriate power consumption estimation. There are several assumptions in the estimation to be discussed with and it is also necessary to evaluate ULPDDP performance from several aspects. Non-UDP protocol performance is one of the important areas for mobile network application of ULPDDP.

As it is described in the paper, in the previous study, because of using low error rate of data transfer network and easy implementation, re-transmission effect to data driven networking system has not yet been evaluated. However, error and re-transmission at radio section is regularly occurring issue so that it plays an important role in the mobile communications performance. Further study on this area and detailed evaluation is required. Also further work is expected for implementation of mobile protocols on ULPDDP with

fully utilizing mobile communication parallelism described in this paper. After the work, an experiment is expected to accurate power consumption comparison with DSP implementation.

Another further work is expansion of application of ULPDDP in the wireless base station. FPGA and ASIC are employed signal processing of Channel Unit to reduce DSP load and realize high speed signal processing. These functions such as coding /decoding and FFT are pipeline process and have good nature applying data driven scheme. Studying to apply STP to such function is expected as good start and is beneficial to develop low power channel unit. It will expand possibility of software control or software implementation of these functions together with ULPDDP.

Studying heterogeneous architecture such a mixture of ULPDDP and other co-processor function of mobile signal processing is also expected useful for future expansion of ULPDDP application.

Another challenge is software development for the implementation. DSP already has development environment and software asset. It is expected to enrich ULPDDP environment for easy introduction and implementation

4 Acknowledgement

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.

5 References

- [1] Hidehiko Ooyane, Daisuke Tanigawa, Naoki Nakaminami and Yoshitaka Hiramoto, "Development of IP-based wireless base transceiver station," NTT DoCoMo Technical Journal, Vol. 15, No. 1, Apr. 2007
- [2] CPRI Specification V4.2, Sept. 2010
- [3] Vodafone Chair Mobile Communications Systems, "Study on Energy Efficient Radio Access Network (EERAN) Technologies", Project Report of TU Dresden, 2009
- [4] 3GPP TR 32.826 V2.0.0, "Study on Energy Savings Management (ESM)", Mar. 2010
- [5] Shuji Sannomiya, Ryotaro Kuroda, Kazuhiro Aoki, Kei Miyagi, Makoto Iwata and Hiroaki Nishikawa, "Chip-Multi Processor 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, PDP5136, July 2011
- [6] K. Miyagi, S. Sannomiya, K. Sakai, M. Iwata and H. Nishikawa, "Autonomous Power-Supply Control for Ultra-Low-Power Self-Timed Pipeline", PDPTA'08, pp. 704-709
- [7] 3GPP TS 36.300(V8.10.0) Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Overall description; Stage 2 (Release 8), Sept. 2009
- [8] 3GPP TS 25.301(V8.5.0) Radio Interface Protocol Architecture(Release 8), Mar. 2009
- [9] 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, PDP5135, July 2011

Broadcast Voice Streaming by Load-aware Flooding over Ad Hoc Networks achieving Reduction of Traffic and Power Consumption

Keisuke Utsu^{1,2}, Hiroaki Nishikawa³, and Hiroshi Ishii¹

¹School of Information and Telecommunication Engineering, Tokai University, Minato, Tokyo, Japan

²Research Fellow of the Japan Society for the Promotion of Science, Japan

³Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki, Japan

Abstract - 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. For the broadcast delivery of high-load information flows such as voice streams, we have already proposed a Load-aware Dynamic Counter-based Flooding (LDCF). In this paper, LDCF is applied to broadcast voice streaming, and is compared with SF using network simulation. It is shown that LDCF results in reduced packet loss and a lower volume of generated traffic than SF.

Keywords: Ad Hoc Network, Voice, Streaming, Flooding, Broadcast

1 Introduction

Ad hoc networks [1] are likely to be used in a disaster situation because these wireless networks that do not depend on an established radio infrastructure. In a disaster it is likely that a limited number of specific nodes will broadcast voice streams to the entire network. For example, evacuation instructions or information about damage suffered may be broadcast to the wireless mobile terminals of disaster victims. In addition, instructions may be sent to rescue teams. However, unicast or multicast communication based on the conventionally studied routing protocols cannot be applied to the above situation for the following reasons. First, as the number of nodes increases, so does the data flow, resulting in an increase in traffic. Second, since these techniques require some time before new routing tables are established, they cannot immediately deliver information. Third, their routing is based on the knowledge of the IP addresses of other nodes, an assumption not likely to be fulfilled in a disaster situation. For these reasons, the only viable alternative is to broadcast packets to the entire network, just as conventional flooding does. However, if simple flooding (SF) in [2] 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, and many unnecessary packets will be broadcast, resulting in degradation in quality of service and an increase in traffic and power consumption. There have been no previous studies that focused on the application of broadcast voice streaming to an ad hoc network.

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 [3]. However, we did not evaluate QoS, the reduction effectiveness of traffic and power consumption per node for the case in which voice streaming is used. This paper shows 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. This paper shows that LDCF generates less traffic and consumes less power per node than SF.

Section 2 describes the assumed environment and the proposed flooding method, LDCF. Section 3 describes an evaluation of the average packet loss rate, the volume of traffic and the estimated power consumption per node using network simulation. Section 4 presents the conclusions.

2 Flooding methods

2.1 Existing methods and their problems

Simple Flooding is used to deliver packets to the whole network as follows. A node that originates video 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. Since packets are generated at a very high rate in a voice streaming, SF produces many redundant re-broadcasts, which can cause many collisions and buffer overflows,

resulting in packet loss. Thus, it is difficult to maintain an adequate and stable level of communication quality. In particular, if there is a dense population of nodes in the network, communication quality can be seriously degraded.

There have been proposals to improve SF in order to reduce redundant re-broadcasts, thereby raising the probability of the packets reaching many nodes [4-5]. Counter-based scheme in [4-5] is one of the most famous improved flooding methods without using extra facility such as GPS (Global Positioning System). The method makes the rebroadcast decision based on the number of receiving same packets. Basic algorithm is as follows. When a node receives a packet the counter (*counter*) is initialized. When the node receives the same packets during a random period again, the counter is added 1. If the counter reaches the counter threshold value (*c_threshold*), the rebroadcast is canceled. Otherwise (the timer expires without reaching *c_threshold*), the node rebroadcasts the packets. The *c_threshold* affects the performance of this method [4]. In a sparse network, if the *c_threshold* is small (such as 2), rebroadcast is restrained, but low packet reachability is achieved. On the other hand, in a dense network, the *c_threshold* does not affect the packet reachability seriously. The author of [4] recommends that a *c_threshold* of 3 or 4 is probably a reasonable choice. In addition, when a large *c_threshold* (such as 6) is selected, the behavior is close to Simple Flooding, so the redundant rebroadcast is not restrained. To improve this method, the Adaptive Counter-based scheme [6] has been proposed. In this improved method, nodes transmit Hello packets periodically and recognize the number of neighboring nodes to decide the *c_threshold* dynamically. However the periodical transmission of Hello packets is not favorable in terms of energy consumption and wireless resources efficiency.

Assuming the broadcast video streaming, the above methods have critical problems. When packets are generated at a high rate in the network, the nodes cannot avoid traffic congestion because these methods consider only redundancy of the same packets. This can lead to degradation in communication quality, and an accelerated consumption of power by these nodes.

2.2 LDCF (Proposed method)

As mentioned in Section 2.1, several improved flooding methods have been proposed. However none of them are applicable to broadcast video streaming. Since the methods in [3-6] are not load-aware methods, to achieve broadcast video streaming with high QoS performance achieving reduction of traffic and power consumption, it is necessary to take into account of load condition at each node. In addition, the method in [6] needs the periodical transmission of Hello packets, which is not favorable in terms of energy consumption and wireless resource efficiency. Hence, we have proposed a novel flooding method taking into account of

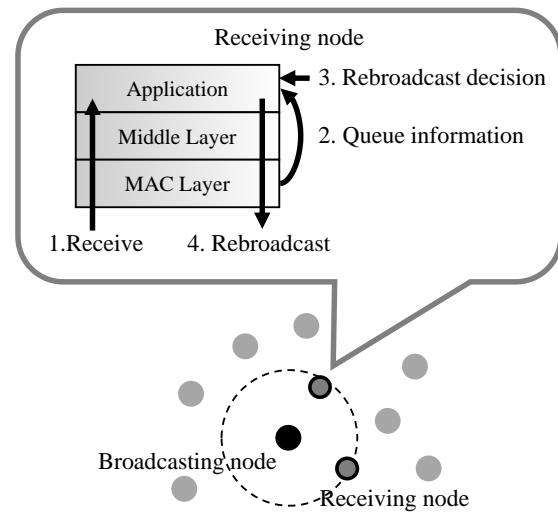


Fig.1 Basic concept of our proposal

load condition at each node without the Hello packet transmission and special equipment such as GPS. In the following, we explain the concepts and operations of our methods.

This paper focuses on unidirectional live streaming, and assumes the following. Nodes in the network are ordinary laptop PCs not equipped with any extra facility, such as GPS, and they communicate with each other using wireless LANs that are based on IEEE802.11. The network uses UDP/IP, performs no QoS control, and operates on the principle of best effort. One video frame can be included in one packet. The paper focuses on the evaluation of video delivery performance, and thus does not delve into video compression methods or the reconfiguration of video frames.

To solve the problems of the existing methods, we have proposed Load-aware Dynamic Counter-based Flooding (LDCF) [3]. Our proposal has following two characteristics: (i) The packet rebroadcast criteria can reflect the actual load condition of each node, and (ii) No Hello packets are sent to get/take network load condition to/from the other nodes. Figure 1 shows the basic concepts of our proposed method. In this scheme, a node that has received a packet learns about its own loading condition by examining the number of packets existing in its MAC queue, and depending on the loading condition, dynamically changes the criterion by which it decides whether to re-broadcast packets. The reason why our method uses the number of packets in the queue because at the time the packets are existing in a node's queue, the node is transmitting/receiving frames exceeding its link capacity. If the node rebroadcasts a packets 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. A detailed operation of this scheme is described below. Figure 2 shows its algorithm.

The node that has received a packet from an initiator node operates as follows:

```

o Parameter Integer: default_c_threshold
// The default threshold value of the counter.
o Parameter Integer: loaded_c_threshold
// The threshold value of the counter for loaded-nodes.
o Parameter Integer: q_threshold
// The threshold number of packets on the queue.
o Parameter Integer: factor // The factor for loaded-nodes.
o Variable Integer: c_threshold // The threshold value of the counter.
o Variable Integer: counter
o Variable Integer: queue
o Variable Real: decision_time
o Function: getQueue()
// The function to get the number of packets on the queue.

Receive a packet
  if (The same packet as that the node has already received)
  then END
  else if (TTL == 0)
  then END
  else
    counter ← 1
    queue ← getQueue()
    if (queue ≥ q_threshold)
    then decision_time ← Random()*factor
         c_threshold ← loaded_c_threshold
    else decision_time ← Random()
         c_threshold ← default_c_threshold
    end if
  end if
  while (decision_time)
  if (Receive the same packet again)
  then counter++
       if (counter == c_threshold)
       then END
       end if
  end if
  end while
  Rebroadcast the packet
end if
END

```

Fig. 2 LDCF (Operation of receiving node)

1. A node that has received a packet. If the packet has NOT been already received, it sets 1 to the counter (*counter*).
2. It examines the number of packets existing in its MAC queue.
 - 2.1. If the number of packets existing in the MAC queue is equal to or greater than the queue threshold (*q_threshold*), which has been pre-set for that node, the counter threshold (*c_threshold*) is set to *loaded_c_threshold*. The waiting time before the decision as to whether to re-broadcast packets is made (*decision_time*) is expressed as $\text{Random()} * \text{factor}$.
 - 2.2. If the number of packets existing in the MAC queue is smaller than the queue threshold (*q_threshold*), which has been pre-set for that node, the counter threshold (*c_threshold*) is set to *default_c_threshold*. The waiting time before the decision as to whether to re-broadcast packets is made (*decision_time*) is expressed as $\text{Random}()$.
 - 2.3. If the node re-receives the same packet before *decision_time* has elapsed, it adds 1 to counter. If the counter value reaches *c_threshold*, the packet is not rebroadcast. If the counter value has not reached *c_threshold*, the packet is sent to the lower layer, and it is re-broadcast.

Table 1 Configuration of the simulation

	Simulation area [m]	Initiated packet size [byte]
Case A	1000x600	200
Case B	2000x1200	200
Case C	1000x600	60
Case D	2000x1200	60

Table 2 Parameter of LDCF

	Cx_n1	Cx_n5	Cy_n1	Cy_n5	Cz
<i>default_c_threshold</i>	6	6	4	4	2
<i>loaded_c_threshold</i>	2	2	2	2	1
<i>n</i>	1	5	1	5	-

It is assumed that $\text{loaded_c_threshold} < \text{default_c_threshold}$, and $\text{factor} = 2^n$, this has been introduced to increase the probability of avoiding the collision of data frames in loaded nodes. (A detailed description of *factor* is omitted in Fig. 1.) Here, *n* is assumed the integer between 0 and 5. The domain of definition is such that it allows a sufficient delay time to ensure the delivery of a packet to the entire network. This is because in this paper it is assumed that the network is so small that all nodes can be reached within around 5 hops.

3 Evaluation using simulation

Although we have indicated that LDCF is suitable for high-load streaming, we have not studied its use in broadcast voice streaming. In this section, using a network simulator [7], LDCF is applied to broadcast voice streaming, and the QoS of this streaming communication is evaluated. LDCF is compared with SF in terms of the packet loss rate and the volume of traffic per node.

3.1 Simulation Conditions

The simulation conditions were as follows. There were 100 nodes in the network, of which one to six nodes originated voice streams (initiator nodes). All nodes moved around based on the Random Waypoint Model at a speed ranging from 0.00 to 8.00 m/s, which simulates the walking speed of humans. The MAC layer of each node used IEEE802.11b. The data bit-rate was 2 Mbps. The transmission power was 0.005 W. The received power for successful packet reception was -85 dBm.

As shown in Table 1, Two simulation areas were considered: 1000 m x 600 m (Cases A and C) and 2000 m x 1200 m (Cases B and D). Two packet sizes (L2 payload) were used: 200 Bytes (Cases A and B), which assumed the use of G.711 [8] for coding and decoding, and 60 Bytes (Cases C and D), which assumed the use of G.729 [9]. Packets were

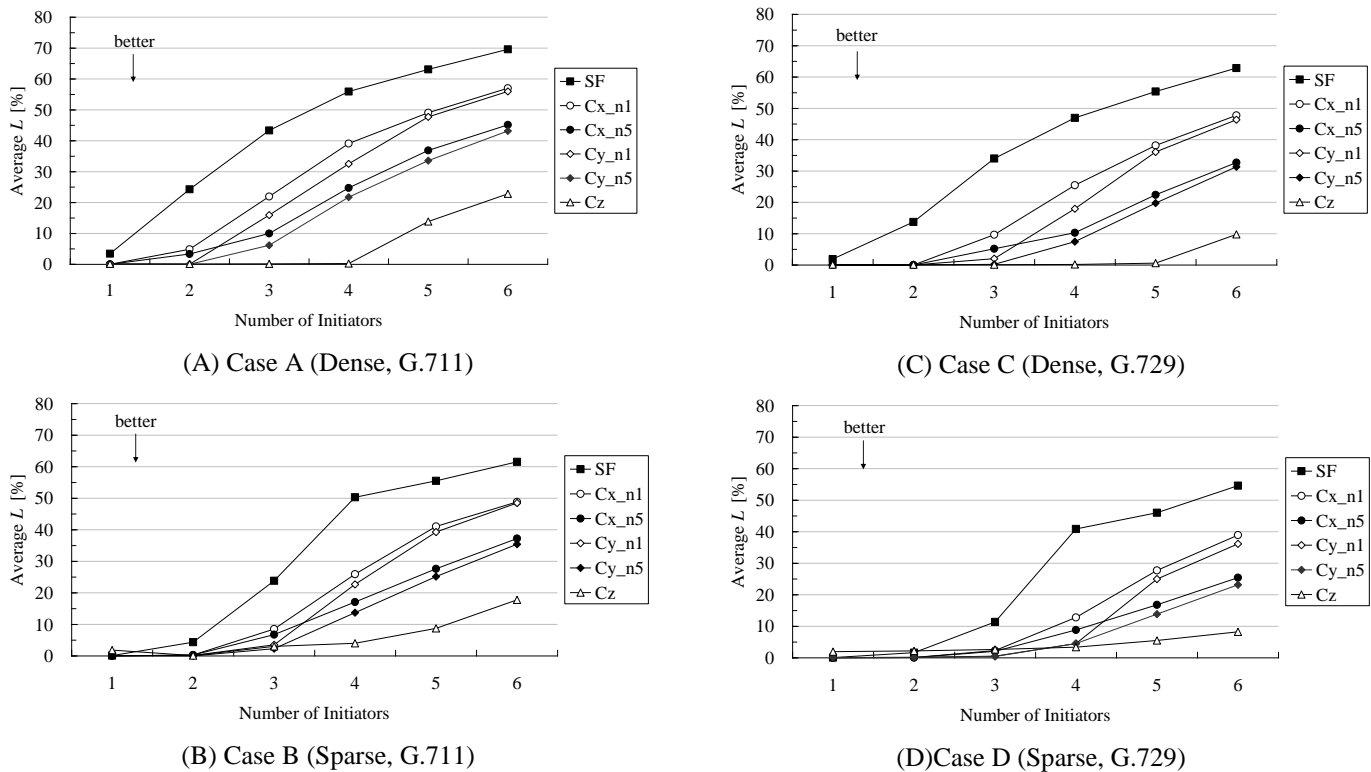


Fig.3 Simulation results for average packet loss rate, L , at candidate nodes for different sets of parameter

generated at intervals of 20 ms. Each initiator node originated 1500 packets, i.e., a 30-second-long voice stream. Either LDCF or SF was used to broadcast packets generated by the initiator nodes. One of the 5 sets of parameter values shown in Table 1 was used for LDCF. In the case of Cz, $loaded_c_threshold = 1$. Therefore, packets were not re-broadcast if $queue \geq q_threshold$. This meant that the number of initiator nodes, n , did not affect the operation of LDCF. Initiator nodes did not re-broadcast packets they received from other nodes. The play-out deadline at a receiving node was 5 seconds. Any packets that arrived after the play-out time were discarded.

We conducted 20 simulation trials for each Case, for each set of parameter values, for different location of nodes and mobility patterns.

3.1.1 QoS Evaluation

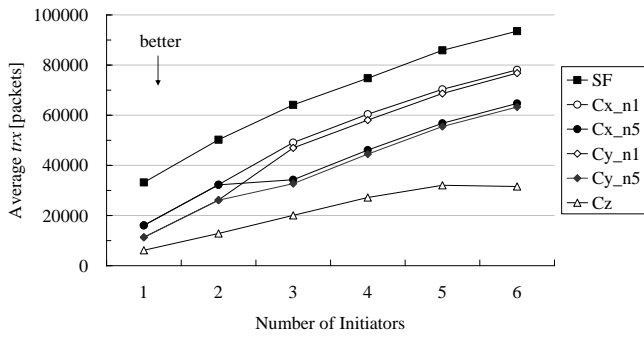
Specific measures used for the evaluation of the QoS of Voice over IP (VoIP) included packet loss, packet delay, and delay jitter [10][11]. However, since we were focusing on unidirectional live voice streaming in the situation of a disaster as mentioned in Section 2.1, several seconds of delay in reproducing voice are tolerable. The problems of packet delay and packet jitter can be solved by buffering packets.

Therefore, only packet loss needs to be examined in our evaluation. We examined the average packet loss rate, i.e., the ratio of the number of successfully received packets to the number of packets generated, as follows. In each trial, 5 nodes

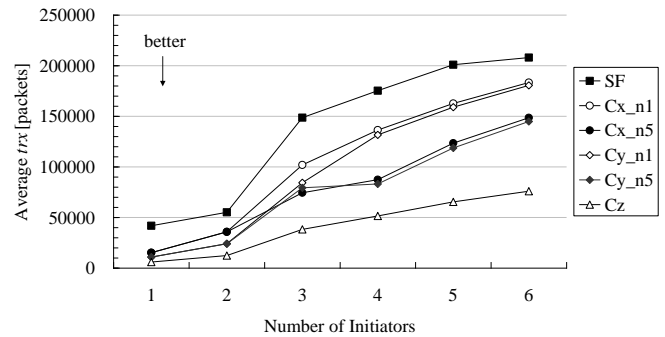
were selected for observation. From among all the packets generated by an initiator node, those that were successfully received were counted. This was repeated for 20 simulations. We thus obtained samples of 100 observed nodes. The average packet loss rate, L [%] was the average for the 20 simulations.

The evaluation results are shown in Fig. 3. In all Cases (Cases A to D), L was smaller when LDCF was used, for all sets of parameter values, than when SF was used. The reason is that LDCF reduced unnecessary re-broadcasting, and as a result, reduced the occurrence of collisions and buffer overflows. The value of L was the smallest with parameter values Cz. This is because the smaller $default_c_threshold$ and $loaded_c_threshold$ are, the more greatly was unnecessary re-broadcasting reduced.

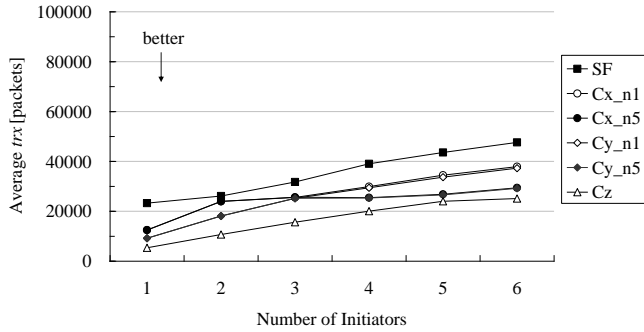
Next, we examine QoS and the scalability of LDCF. According to Reference [10][12], when G.711 and PLC are used, the MOS (Mean Opinion Score) is greater than 2 at a packet loss rate of 5%, irrespective of whether packet losses occur at random or are bursty. An MOS of 2 means Poor while that of 3 means Fair. If broadcast voice streaming is to be used in a disaster situation for sending evacuation instructions or information about damage, some degradation in QoS can be tolerated as long as listeners can understand what is said by the speaker. Therefore, broadcast voice streaming is considered useful if $L < 5\%$. In Cases C and D, a G.729 codec was used. The MOS of a G.711 codec by itself is 4.11 while that of a G.729 codec is 3.92 [11]. The difference



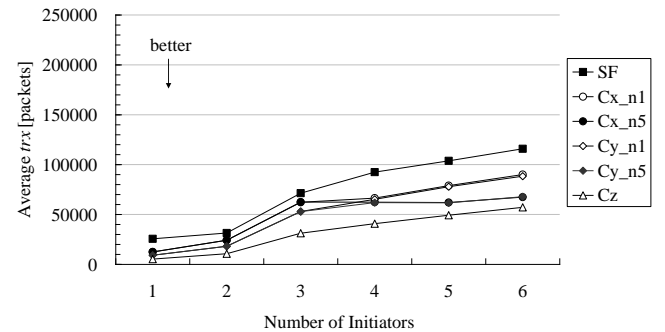
(A) Case A (Dense, G.711)



(C) Case C (Dense, G.729)



(B) Case B (Sparse, G.711)



(D) Case D (Sparse, G.729)

Fig.4 Simulation result of average total traffic per node *trx* for different set of parameters

in MOS between G.729 and G.711 codecs is so small that we assumed that broadcast voice streaming was useful even in Cases C and D as long as $L < 5\%$.

For LDCF, we focused on parameter values Cz, with which L was the smallest. When there were 4 initiator nodes, L was 0.23% in Case A, 3.89% in Case B, 0.18% in Case C, and 3.05% in Case D. In other words, all Cases satisfied $L < 5\%$, suggesting that broadcast voice streaming can provide sufficient QoS even under the assumed conditions as long as the number of initiator nodes is no greater than 4.

3.1.2 Evaluation of the volume of traffic per node

We examined the volume of traffic per node. The specific items examined were (i) to (iii) below. These were calculated for each simulation.

(i) Number of transmitted packets per node, *trs*

The number of packets that were received and then re-broadcast at each node was totaled for all the nodes in the network, and the sum was then divided by the number of nodes. The packets originated by initiator nodes were excluded from the above count. Only re-broadcast packets were counted.

(ii) Number of received packets per node, *trr*

The number of packets that were received at each node was totaled for all the nodes in the network, and the sum was then divided by the number of nodes.

(iii) Number of transmitted and received packets per node, *trx*

trx is the sum of *trs* and *trr*.

Figure 4 shows the average *trx* for all trials. In all Cases, *trx* was smaller with LDCF for any of the sets of parameter values than with SF. The reason is that LDCF reduced unnecessary re-broadcasting, and as a result, reduced the number of received packets. In the case of set Cz, which gave the smallest L , and 4 initiator nodes, *trx* was reduced from that of SF by 63.7% in Case A, 48.4% in Case B, 70.6% in Case C, and by 55.5% in Case D.

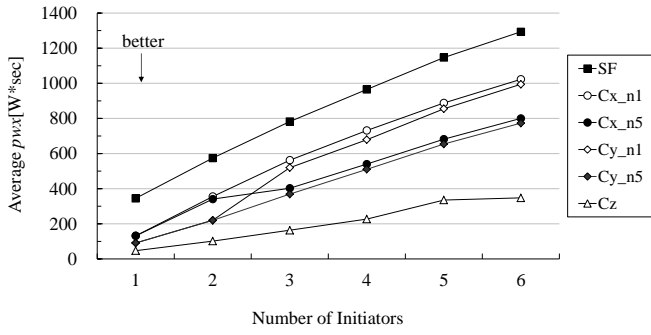
3.1.3 Evaluation of estimated power consumption

Based on the results obtained in Section 3.1.2, the power consumption per node was calculated for different sets of delivery parameters, and the results were compared. According to a study by Freney et al. [13], the energy consumed when one packet is transmitted, $tp[\mu W \cdot sec]$, and that when one packet is received, $rp[\mu W \cdot sec]$, can be expressed as:

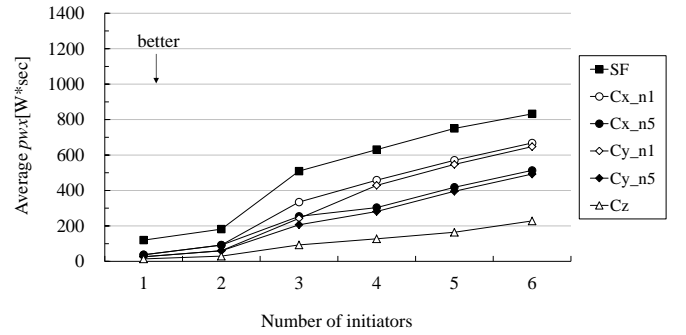
$$tp = 2.000 * frame\ length + 270 \quad (1)$$

$$rp = 0.500 * frame\ length + 60 \quad (2)$$

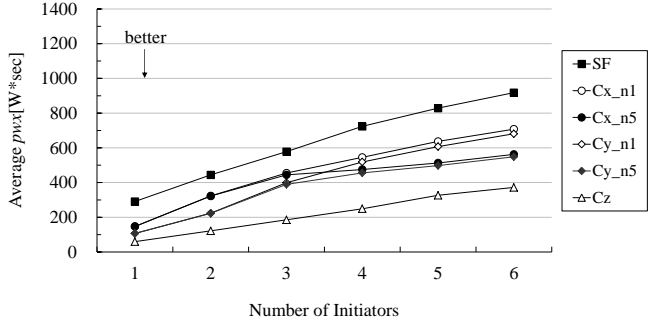
Using expressions (1) and (2), the following can be calculated.



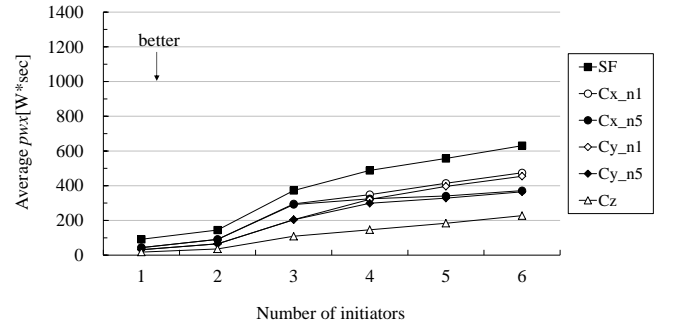
(A) Case A (Dense, G.711)



(C) Case C (Dense, G.729)



(B) Case B (Sparse, G.711)



(D) Case D (Sparse, G.729)

Fig.5 Simulation result of average total traffic per node pwx for different set of parameters

(iv) Power consumption of packet transmission per node, pws [$\mu W \cdot sec$]

$$pws_{param} = trs_{param} * tp \quad (3)$$

(v) Power consumption of packet reception per node, pwr [$\mu W \cdot sec$]

$$pwr_{param} = trr_{param} * rp \quad (4)$$

(vi) Power consumption of packet transmission and reception per node, pwx [$\mu W \cdot sec$]

$$pwx_{param} = pws_{param} + pwr_{param} \quad (5)$$

Figure 5 shows the average pwx for all trials. In all Cases, pwx was smaller with LDCF for any of the sets of parameter values than with SF. The reason is that LDCF reduced unnecessary re-broadcasting, and as a result, reduced the number of received packets. In the case of set Cz, which gave the smallest L , and 4 initiator nodes, pwx was reduced from that of SF by 76.5% in Case A, 65.6% in Case B, 79.8% in Case C, and by 70.0% in Case D.

4 Conclusions

In this paper, we have studied the application of LDCF to broadcast voice streaming in an ad hoc network, and evaluated the effects of LDCF using network simulation.

To evaluate QoS, we examined the average packet loss rate. It was found that the average packet loss rate was smaller

with LDCF than with SF. In the case of parameter value set Cz, which used the lowest counter threshold, broadcast voice streaming provided sufficient quality as long as the number of initiator nodes was no greater than 4. We also evaluated the volume of traffic per and estimated power consumption. The result showed that these values were smaller with LDCF than with SF.

In this paper, we used the average packet loss rate to evaluate QoS. To evaluate QoS in more detail, we plan to use objective testing methods, such as perceptual evaluation of speech quality (PESQ) [14]. We also plan to study the use of LDCF for broadcast video streaming and mixed voice and video streaming.

5 Acknowledgments

This work was supported by JSPS KAKENHI (Grant-in-Aid for JSPS Research Fellows) and Core Research for Evolutional Science and Technology (CREST), Japan Science and Technology Agency (JST).

6 References

- [1] C. Siva Ram Murthy, 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] Keisuke Utsu, Hiroshi Ishii, "Load-aware Flooding for Streaming over Ad Hoc Networks", IEEJ Trans. EIS, Vol.130, No.8, 2010
 - [4] Yu-Chee Tseng, Sze-Yao Nis, Yuh-Shyan Chen, Jang-Pinig Sheu, "The Broadcast Storm Problem in a Mobile Ad Hoc Network," Wireless Networks Volume 8, Springer, pp.153-167, Kluwer Academic Publishers, Mar. 2002
 - [5] Brad Williams and 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, Jun. 2002
 - [6] 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
 - [7] The network simulator "OPNET", <http://www.opnet.com>
 - [8] ITU-T Recommendation G.711, "Pulse code modulation (PCM) of voice frequencies"
 - [9] ITU-T Recommendation G.729, "Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)"
 - [10] J.H. James, Bing Chen, Laurie Garrison, "Implementing VoIP: A Voice Transmission Performance Progress Report", IEEE Communications Magazine, Vol. 72, Issue. 7, pp.36-41, July 2004
 - [11] Stylianos Karapantazis, Fotini-Niovi Pavlidou, "VoIP: A comprehensive survey on a promising technology", Computer Networks, 53, 2009, pp.2050-2059, Elsevier
 - [12] ITU-T Recommendation P.800, "Methods for subjective determination of transmission quality"
 - [13] 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. Proceedings, IEEE Volume 3, April 2001, pp.1548-1557
 - [14] ITU-T Recommendation P.862, "An objective method for end-to-end speech quality assessment of narrow-band telephone networks and speech codecs"

Proposal on Battery-aware Counter-based Flooding over Ad Hoc Networks

Keisuke Utsu^{1,2}, Hiroshi Sano¹, Turganzhan Kassymov³, Hiroaki Nishikawa⁴, and Hiroshi Ishii¹

¹School of Information and Telecommunication Engineering, Tokai University, Minato, Tokyo, Japan

²Research Fellow of the Japan Society for the Promotion of Science, Japan

³Graduate School of Engineering, Tokai University, Hiratsuka, Kanagawa, Japan

⁴Graduate School of Systems and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki, Japan

Abstract - 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. This paper considers cases where a specific number of nodes in an ad hoc network broadcast video and audio streams over the entire network, and proposes Battery-aware Counter-based Flooding (BCF), which reduces degradation in delivery quality, and energy consumption, and prevents nodes from being interrupted due to the complete discharge of their batteries. It also describes an evaluation of this flooding scheme using network simulation, and confirms its effectiveness.

Keywords: Ad Hoc Network, Broadcast, Flooding, Battery, Low power consumption

1 Introduction

As happened in the recent earthquake in Japan (Northern Japan Earthquake), a large-scale disaster can disrupt existing communications infrastructures. As a solution to this problem, a network configuration known as an ad hoc network is being studied [1]. Conceived applications of an ad hoc network during a disaster include real-time streaming delivery, from a specific number of nodes to the entire network, of video and audio data that show scenes of disasters or instructions on evacuation.

Most studies on ad hoc networks have considered client-server-type unicast or multicast streaming delivery. Some have focused on video streaming [2-4]. However, communications based on existing unicast routing protocols are not suitable for delivering data over an entire network that consists of many nodes. They require appropriate IP addresses to be allocated to all the nodes in the network before the delivery of data, a condition it may be difficult to meet in a disaster. One method of delivering data without using a routing protocol is flooding, which broadcasts data. However, if the existing Simple Flooding [5] is used to deliver streaming video and audio data, which generates packets at a high rate, redundant re-broadcasts can occur, resulting in an increased chance of data frame collisions and buffer

overflows, and thereby considerably degrading the delivery quality. It is necessary to study how to achieve high packet reachability. There have been some proposals for revising Simple Flooding to reduce redundant re-broadcasts [6-8].

In a disaster, it is often difficult to ensure a stable supply of power to communications terminals and base stations. Although an ad hoc network is not dependent on control by base stations, its terminals are dependent on a finite battery charge. 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 (i) to achieve high packet reachability and reduce the degradation of delivery quality, (ii) to reduce the energy consumption of nodes to make effective use of their battery charge, and (iii) to reduce the chances of nodes becoming inoperative due to the complete discharge of their batteries. However, there have been no studies that attempt to satisfy all these requirements simultaneously. In the light of this, this paper, proposes Battery-aware Counter-based Flooding (BCF), evaluates it using network simulation, and confirms its effectiveness.

Section 2 identifies problems with existing methods. Section 3 describes the proposed method. Section 4 shows the effectiveness of the proposed method using network simulation and Section 5 gives the conclusions.

2 Existing methods and their problems

Simple Flooding (SF) [5] is the most frequently used broadcast-type delivery. 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 [6-8]. A representative one among them is as follows.

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 [6-7].

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* [6]. 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 message periodically. These messages 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 [9-10]. Koide et al. [9] 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 [9] 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 [10] 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 message packets, should not be used, in order to reduce the network load and power consumption by nodes.

3.2 Battery-aware counter-based flooding (BCF)

To meet the requirements listed in Section 3.1, this paper proposes a flooding method that is sensitive to the remaining battery levels of nodes and uses a counter in determining whether a packet should be re-broadcast or not. We call this scheme Battery-aware Counter-based Flooding (BCF). Requirement (i) is met as follows. As is used in existing counter-based schemes, BCF 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. BCF 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 BCF 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:

$$c_threshold = \text{ceil}(max_c_threshold * (remain_battery / max_battery)) \quad (1)$$

where $\text{ceil}(\text{Real } x)$ 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 for $decision_time$. If during this time it receives the same packet again during the waiting time, it adds "1" to counter.
4. If, at the end of $decision_time$, counter exceeds $c_threshold$, the received packet is not re-broadcast but discarded. If counter does not exceed $c_threshold$, the received packet is sent to the lower layer and re-broadcast.

4 Evaluation using network simulation

This section compares the performance of the proposed BCF with those of existing schemes, using a network simulator, OPNET [11].

4.1 Simulation conditions

It is assumed that there are 100 nodes in the network, and that one of them is the initiator node. Four cases are considered. In Cases A and B, the initiator node generates

Variables and parameters

- 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 remaining battery level.
- Function: $getBattery()$
// The function to get the remaining battery level.

The method of getting data on the remaining battery level

Executed at certain intervals ($get_interval$)

```
remain_battery ← getBattery()
c_threshold ← ceil(max_c_threshold * (remain_battery / max_battery)).
```

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 (Receive 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 receiving node for BCF

video streams, while in Cases C and D, it generates audio streams. We evaluate BCF and compare it with other schemes in terms of the extent to which power is saved, the extent to which the number of nodes that have become inoperative due to complete discharge of their batteries is reduced, and the delivery performance. We disregard play-out control of the video/audio stream by the receiving node, and focus on the packet reachability at the network layer. Specific details of the video and audio streams used are as follows.

(i) Evaluation with video streams

The use of H. 264 codecs [12] is assumed. The initiator node generates video streams as follows. On the "highway" [12] that uses the Quarter Common Intermediate Format (QCIF), 1,000 frames are encoded using jm14.2 [13]. 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(a) shows the packet size distribution for 1,000 initiated packets. The initiator node repeats the transmission of these 1,000 frames 36 times, so generating 36,000 packets, which is equivalent to a 20-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.

(ii) Evaluation with audio streams

The use of G. 711 codecs [14] is assumed. The size of a packet generated by the initiator node is fixed at 200 Bytes, as shown in Figure 2(b). Packets are generated at intervals of 20 ms. The initiator node generates 36,000 packets, equivalent to a 12-minute-long audio stream. To take account of the rate at which packets are generated, the value generated by Random() at each node is set to [0, 20] ms. 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 BCF. 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: C2, C3 and C4. Similarly, using different values of $max_c_threshod$, three cases are considered for BCF: B2, B3 and B4. In BCF, the interval at which the remaining battery level is monitored ($get_interval$) is 10 seconds.

Two geographical network areas are considered for simulation. In Cases A and C, the area is 1000 m x 600 m, representing a network in which nodes are densely distributed. In Cases B and D, the area is 2000 m x 1200 m, representing a network in which nodes are sparsely distributed. 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.

The remaining battery level of a node is simulated as follows. The maximum battery level ($max_battery$) is 200 W*s. At the start of the simulation, the initial battery level of the initiator node is 200W*s (i.e., 100% full), and those of other nodes are random in the range [40, 160] W*s (i.e., 20% to 80% full).

Feeney et al. [15] have indicated that the transmission power, tp [μ W*sec], and the reception power, rp [μ W*sec], for a single packet are as follows:

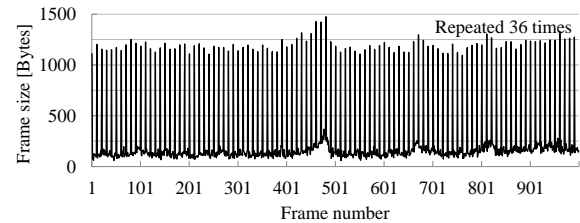
$$tp = 2.000 * frame\ length\ [byte] + 270 \quad (3)$$

$$rp = 0.500 * frame\ length\ [byte] + 60 \quad (4)$$

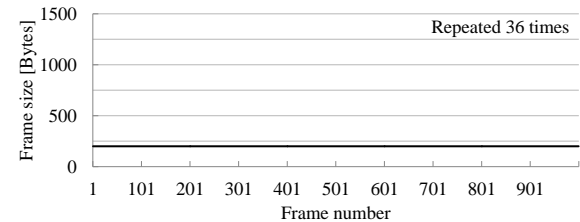
We assume that the above power is consumed by a node each time it sends or receives a packet. Any power that may be

Table 1 Parameters for each delivery method

	$c_threshold$	$max_c_threshold$
SF	-	-
C4	4 (fixed)	-
C3	3 (fixed)	-
C2	2 (fixed)	-
B4	Determined by Eq.(2)	4
B3	Determined by Eq.(2)	3
B2	Determined by Eq.(2)	2



(a) Cases A and B (assuming a H.264 encoded video)



(b) Cases C and D (assuming a G.711 encoded voice)

Fig. 2 Size distribution of initiated packets

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 Evaluated 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.

(a) 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.

(b) 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.

(c) 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.

(d) Average packet reachability [%] and average delivery time [s]

Packet reachability is the percentage of the nodes that have received 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. The packet delivery time is the time that elapsed from the time when the application layer of the initiator node generated a packet to the time when the application layer of a node successfully received that packet. The average delivery time is the average for all the packets generated by the initiator node. The smaller this value, the better.

5 Evaluation results

5.1 Evaluation for video streams

(a) Average energy consumed by a node till the end of the streaming

The evaluation result is shown in Fig. 3(a). The deviation bars show the (average +/- standard deviation). In Case A, B2 to B4 consumed less energy than the existing schemes. In particular, B2 consumed the least power, 59.4% down from SF, and 33.7% down from C2 in Case A. In Case B, B2 and B3 consumed less power than the existing schemes. The energy consumption in B2 is 67.2% down from SF and 38.7% down from C2.

(b) Average remaining battery level at the end of the streaming

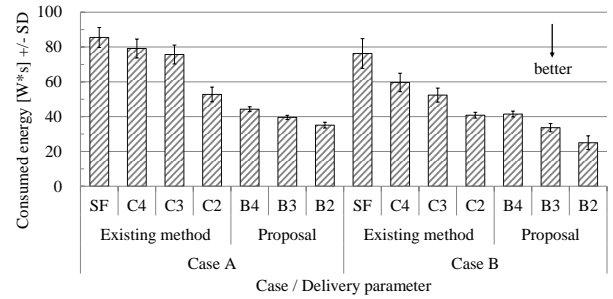
The evaluation result is shown in Fig. 3(b). B2 to B4 registered higher values than any existing scheme in both Cases A and B. In particular, B2 produced the highest value in both Cases A and B. These results indicate that the proposed scheme can preserve a higher battery level than the existing schemes.

(c) Percentage of nodes whose operation was stopped due to complete battery discharge

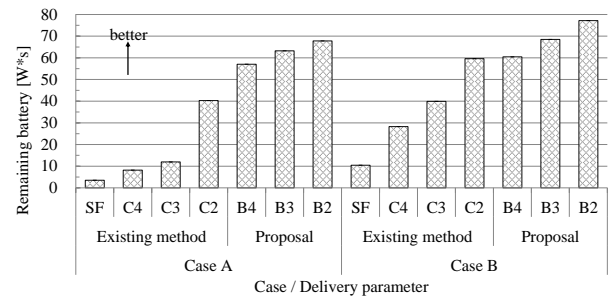
The evaluation result is shown in Fig. 3(c). The deviation bars show the (average +/- standard deviation). B2 to B4 produced a lower percentage than any existing scheme in both Cases A and B. These results indicate that the proposed scheme results in fewer cases in which nodes become inoperative due to complete discharge of their batteries than the existing schemes.

(d) Average packet reachability [%] and average delivery time

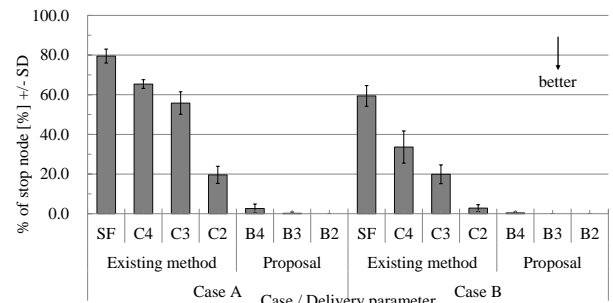
The evaluation result is shown in Fig. 4(d). B4 produced higher average packet reachability than any existing scheme in both Cases A and B. In Case B, the average packet reachability values of B2 and B3 are lower than that of C2,



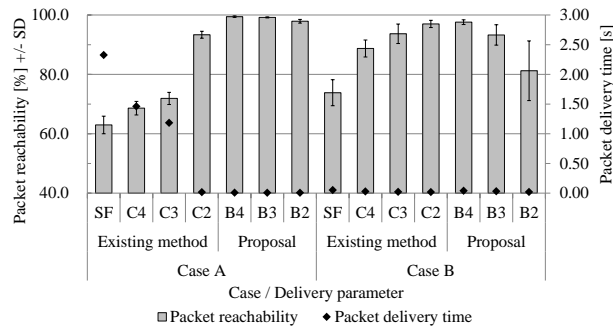
(a) Average energy consumed from start to end of streaming



(b) Average remaining battery level at the end of streaming



(c) Percentage of nodes stopped due to complete battery discharge



(d) Packet reachability and delivery time

Fig.3 Simulation results for the evaluation assuming broadcast video streaming (Cases A and B)

which registered the highest value among the existing schemes. This implies that the average packet reachability of the proposed scheme can fall considerably when nodes are sparsely distributed in the network. The average values of packet delivery time of B2 to B4 are extremely short in both Cases A and B.

5.2 Evaluation for audio streams

(a) Average energy consumed by a node till the end of the streaming

The evaluation result is shown in Fig. 4(a). The average values of energy consumed in B2 to B4 in Case C, and in B2 and B3 in Case D, are smaller than those of the existing schemes. In particular, B2 consumed the least power, 52.4% down from SF and 33.7% down from C2 in Case A, and 67.2% down from SF and 64.7% down from C2 in Case B.

(b) Average remaining battery level at the end of the streaming

The evaluation result is shown in Fig. 4(b). The average remaining battery levels of B2 to B4 in Case C, and B2 and B4 in Case D, are greater than those of the existing schemes. In particular, B2 registered the highest value in both Cases C and D. These results indicate that B2 and B3 can preserve battery charge well.

(c) Percentage of nodes whose operation was stopped due to complete battery discharge

The evaluation result is shown in Fig. 4(c). BCF results in a smaller chance of nodes becoming inoperative due to complete battery discharge than SF, C3 or C4 in both Cases C and D. The avoidance by BCF of re-broadcasts by nodes with a low battery level was effective in reducing power consumption in these nodes.

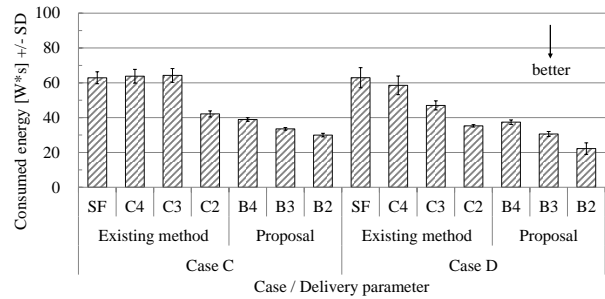
(d) Average packet reachability and average delivery time

The evaluation result is shown in Fig. 4(d). In both Cases C and D, the average packet reachability of B4 was almost the same as that of C2, which registered the highest value among the existing schemes. The difference was less than 1.0%. The average values of packet reachability of B2 and B3 were lower than that of C2 in Case D, indicating that BCF can result in a lower packet reachability when nodes are sparsely distributed in the network. The average values of packet delivery time of SF, C3 and C4 were large in Case C. In contrast, B2 to B4 registered extremely small values in both Cases C and D.

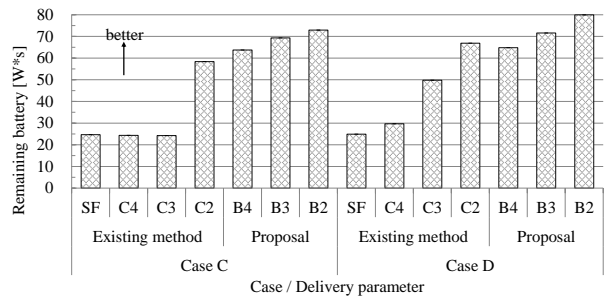
5.3 Discussion

The simulation results described in the above section allow us to conclude the following.

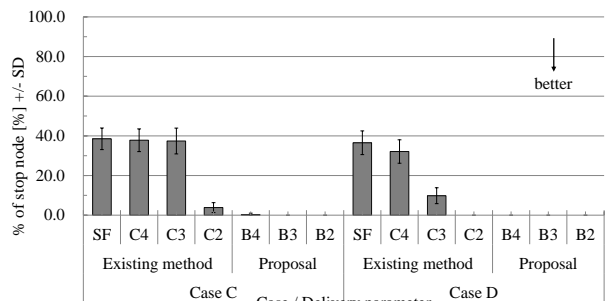
In comparison to existing schemes, BCF can (a) reduce the average power consumed by nodes from the start to the end of the streaming, (b) increase the average remaining battery level at the end of the streaming, and (c) reduce the percentage of nodes which became inoperative due to complete battery discharge. These effects are due to BCF'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-



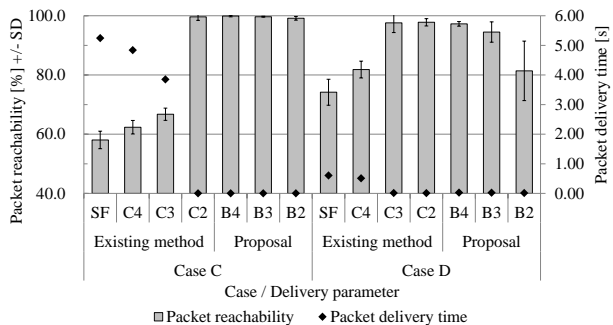
(a) Average energy consumed from start to end of streaming



(b) Average remaining battery level at the end of the streaming



(c) Percentage of nodes stopped due to battery discharging



(d) Packet reachability and delivery time

Fig. 4 Simulation results for the evaluation assuming broadcast voice streaming (Cases C, D)

broadcast packets. B2 produced the best performance in all these three respects, (a) to (c).

The delivery quality was evaluated in terms of (d) the average packet reachability and average delivery time. In networks where nodes are distributed densely (Cases A and C),

BCF produced higher average packet reachability than Simple Flooding or Counter-base schemes with fixed $c_threshold$ (C3 and C4). In networks where nodes are distributed sparsely (Cases B and D), BCF showed higher average packet reachability than Simple Flooding but B2 and B3 gave lower average packet reachability than C2. This is a side effect of the mechanism of avoiding re-broadcasts by nodes with a low battery level. In networks where nodes are sparsely distributed (Cases B and D), a measure that can be taken when higher priority is given to maintaining high packet reachability than to avoiding dissipation of remaining battery charges of nodes is not to set too small a value to $max_c_threshold$. In other words, the use of B4 is preferable. The average delivery time of BCF was extremely small. This can be explained as follows. BCF reduces the number of packet transmissions and receptions, which reduces the load on the MAC layer, which in turn reduces the length of time when nodes are busy.

6 Conclusions

This paper has considered broadcast delivery of video and audio streams in an ad hoc network, and studied how paying attention to the remaining battery levels of nodes can maintain high delivery quality, reduce the power consumed by nodes, and reduce the chances of nodes becoming inoperative due to complete discharge of node batteries. Specifically, we have proposed Battery-aware Counter-based Flooding (BCF), in which 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. BCF has been evaluated using network simulation. The evaluation results have shown that, in comparison to existing data delivery schemes, BCF can reduce power consumption by nodes, reduce the chances of nodes becoming inoperative due to complete discharge of node batteries, and avoid degradation of delivery performance, such as packet reachability. In the evaluation of delivery performance, we focused only on packet reachability and packet delivery time. Looking to the practical application of the proposed scheme, we will evaluate the QoS of played-back video and audio streams.

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] Yifeng He, Ivan Lee, and Ling Guan "Optimized Video Multicasting Over Wireless Ad Hoc Networks Using Distributed Algorithm", IEEE Transactions on Circuits and Systems for Video Technology. Vol. 19, NO. 6, pp. 796- 807, Jun. 2009
- [3] Wei Wei and Avideh Zakhor, "Interference Aware Multipath Selection for Video Streaming in Wireless Ad Hoc Networks", IEEE Transactions on Circuits and Systems for Video Technology, Vol. 19, No. 2, pp. 165-178, Feb. 2009
- [4] Shiwen Mao, Xiaolin Cheng and Y. Thomas Hou, "Multiple Description Video Multicast in Wireless Ad Hoc Networks", Mobile Networks and Applications 11, pp. 63-73, Springer Science, 2006
- [5] 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-mbcst.txt>, 2001
- [6] 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
- [7] 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
- [8] 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.
- [9] 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
- [10] 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. 4m pp. 364-372, 2008
- [11] The network simulator "OPNET", <http://www.opnet.com>
- [12] Patrick Seeling, Frank H. P. Firzek and Martin Reosslein, "Video Traces for Network Performance Evaluation," Springer, 2007
- [13] H. 264/AVC Reference Software (jm14. 2), http://iphone.hhi.de/suehring/tml/download/old_jm/
- [14] ITU-T Recommendation G. 711, "Pulse code modulation (PCM) of voice frequencies"
- [15] 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

SESSION

SYSTEMS SOFTWARE + OS + THREADS + PROGRAMMING MODELS + ARCHITECTURE ISSUES

Chair(s)

TBA

Model Checking Task Sets with Preemption Thresholds

Mitchell L. Neilsen

Department of Computing and Information Sciences
 Kansas State University
 Manhattan, KS, USA

Abstract

Several models have been developed for symbolic schedulability analysis of periodic, preemptive task sets in real-time systems [1, 11]. However, these models cannot be used to analyze task sets with preemption thresholds. In this paper, we present a new model with a two-clocks scheduler that can be used to efficiently analyze periodic task sets with preemption thresholds. The new model can also be used to compute an optimal set of preemption thresholds for a given priority assignment. In particular, it can be used to compute a feasible set of preemption thresholds that are as small as possible.

Keywords: model checking, preemption threshold, real-time system, scheduling theory, worst-case response time

1 Introduction

In classic real-time scheduling theory, tasks are frequently assumed to be periodic. To relax the traditional constraints on task arrival times, automata can be used to model task arrival patterns [1]. Such models are expressive enough to model real-time tasks that are periodic, sporadic, preemptive or non-preemptive, and tasks with additional precedence and resource constraints. Typical real-time scheduling algorithms can be easily generalized to automata. An automaton is schedulable if there exists a scheduling strategy such that all possible sequences of events accepted by the automaton are schedulable. It has been shown that the schedulability analysis problem for such models is decidable [5].

Schedulability analysis can be performed in a manner similar to response time analysis in classical real-time scheduling theory. Real-valued clocks can be used to model task execution and deadlines. The model also includes an automaton which is used to represent the scheduler. Then, a model checker, such as UPPAAL [2, 3], can be used to check for missed deadlines and calculate the worst-case response time

for each task. Similar models can be used to verify the correctness of other distributed algorithms [9, 10].

Many existing real-time models have been analyzed using symbolic schedulability analysis [6, 7, 11]. However, if tasks in a periodic task set are allowed to be assigned preemption thresholds – dynamic priorities used to determine if a running task can be preempted – then these existing models cannot be used. Preemption threshold scheduling is important for real-time sensor networks and real-time embedded systems because preemption thresholds can be used to enable energy and memory efficient scheduling in real-time embedded systems [4, 8]. In a previous paper, we developed a model that can be used to determine if a task sets is feasible if the preemption thresholds are previously assigned [12]. The goal of this paper is to develop new models that can be used to both analyze task sets with preemption thresholds and find an optimal set of preemption thresholds that are minimal.

The rest of the paper is organized as follows: the next section describes the input language for the model checker UPPAAL. Section 3 describes a simple two-clocks model that can be used to correctly test for the schedulability of task sets with preemption thresholds. The model can also be used to derive an optimal set of minimal preemption thresholds. Finally, Section 4 provides a brief summary.

2 Task Model

The focus of this paper is on periodic task sets with preemption thresholds. Each task τ_i is characterized by a 5-tuple of natural numbers denoted $(C_i, T_i, D_i, \pi_i, \gamma_i)$ with $C_i \leq D_i$, where C_i is the run time of task τ_i and D_i is its relative deadline; that is, after task τ_i is released for execution, it should complete within D_i time units. For periodic tasks, the period is denoted by T_i . Each task τ_i has a priority of π_i and a preemption threshold of γ_i . Consider the example shown in Table 1. This task set is the original example from Wang and Saksena's paper [13].

It was used to show that some task sets can be scheduled with preemption thresholds when no fixed priority assignment using purely preemptive or non-preemptive priority assignment will work. If each task has its preemption threshold set equal to its priority ($\gamma_i = \pi_i$ for all i), then purely preemptive scheduling results. On the other hand, if all preemption thresholds are set to the maximum priority, then non-preemptive scheduling is realized. In this way, preemption thresholds can be used to represent a wide range of scheduling strategies ranging from preemptive to non-preemptive.

Table 1. Periodic task set

i	C_i	T_i	D_i	π_i	γ_i
0	20	70	50	3	3
1	20	80	80	2	3
2	35	200	100	1	2

The goal of this paper is to develop a symbolic model to verify various properties and determine if the task set can be scheduled using different preemption thresholds. If the task set can be feasibly scheduled, then the model can also be used to obtain a set of minimal preemption thresholds; that is, preemption thresholds that are as small as possible. Next, we turn our attention to the input language used to develop the model using UPPAAL.

The core of the model input language is timed automata extended with data variables and tasks. Each edge of such extended automata can be labeled with three labels:

1. a guard containing a clock constraint and/or a predicate on data variables,
2. an action which can be an input or output action in the form of $a!$ or $a?$, and
3. a sequence of assignments in the form: $x = 0$ when x is a real-valued clock or $v = E$ when v is a data variable and E is a mathematical expression over data variables and constants.

A location of an extended automaton may be annotated with a task or a set of tasks that will be triggered when the transition leading to the location is taken. The triggered tasks will be put into a task queue, like the ready queue in an operating system, and scheduled to run according to a given scheduling policy. The scheduler should make sure that all task constraints are satisfied in scheduling the tasks in the task queue. To model concurrency and synchronization between automata, networks of automata are constructed in the standard way as in UPPAAL with the annotated sets of tasks on locations unioned [1, 7].

Four types of shared data variables can be used for communication and resource sharing:

1. Tasks can share variables with each other, and shared variables are protected by semaphores.
2. Tasks can read and update variables owned by the automata.
3. Automata can read (but not update) variables owned by the tasks.
4. Automata can share variables with each other.

In this paper, we limit the focus to periodic tasks that are independent and do not share resources. Next, we describe how schedulability analysis can be performed using UPPAAL [1].

A network of timed automaton annotated with tasks is considered as a design model. Given an extended automaton and a scheduling policy, the related schedulability analysis problem is to check whether there exists a reachable state of the scheduler automaton where a task misses its deadline. Such states are called non-schedulable states. An automaton is said to be non-schedulable with the given scheduling policy if it may reach a non-schedulable (ERROR) state. Otherwise, it is schedulable.

Consider the task set shown in Table 1. The highest priority task τ_0 (priority $\pi_0 = 3$) has a deadline of 50 and a period of 70. The lowest priority task τ_2 (priority $\pi_2 = 1$) has a deadline of 100 and a period of 200.

To check if a task set is schedulable, we construct a pair of timed automata – one to generate jobs to be released (called the PERIODIC_TASK automaton) and one to schedule the jobs released (called the SCHEDULER automaton). Then, a model checker is used to check the reachability of a predefined error state in the product automaton of the pair. If the error state is reachable, the task set is not schedulable.

In the next section, we describe a new two-clocks SCHEDULER automaton that can be used to test for the schedulability of task sets with preemption thresholds. It can also be used compute an optimal set of preemption thresholds if such an assignment exists.

3 Two-Clocks Model

The following two-clocks model has fewer transitions than previous models. Also, unlike the encoding of the two-clocks scheduler by Fersman, et al. [7], there are five states in our model instead of four:

- $Idle_i$ - the ready job queue is empty,

- $Busy_i$ - jobs with priority greater than or equal to task i have arrived for execution,
- $Ready_i$ - the job in task i to be checked for feasibility has been released for execution, but has not started executing, and
- $Check_i$ - the job in task i to be checked for feasibility has started executing, and
- $Error_i$ - the checked job in task i missed its deadline.

The additional state, $Ready_i$, is needed because of the different way in which jobs may be preempted before or after being scheduled for execution when using preemption thresholds; after a task has been scheduled for execution, it can only be preempted by tasks with a priority greater than the running task's preemption threshold.

The first automaton, called PERIODIC_TASKS, is designed to model the releases of jobs within each task. The SCHEDULER automaton for the example given above is shown below in Figure 2. The following global declarations are used to specify the constraints shown in Table 1:

```
chan job[3];

const int C[3] = { 20, 20, 35 };
const int T[3] = { 70, 80, 200 };
const int D[3] = { 50, 80, 100 };
const int N[3] = { 40, 35, 14 };
const int PR[3] = { 3, 2, 1 };
const int PRT[3] = { 3, 3, 2 };
```

With periods of 70, 80, and 200, the hyperperiod is 2800. In one hyperperiod, there will be 40 jobs released in task τ_0 , 35 in τ_1 , and 14 in τ_2 . The number of jobs over a hyperperiod are specified in $N[3]$. Finally, the priorities and preemption thresholds are specified in arrays $PR[3]$ and $PRT[3]$, respectively. Local declarations and parameters passed to the SCHEDULER automaton are shown below.

```
SCHEDULER(int P, int PT, int RT, int D, int id)
clock c, d;
int r;
int Cmax = 1;

int gt(int i, int x, int r)
{
  if (PR[i]>x)
    return (r+C[i]);
  else
```

```
    return (r);
}

int ge(int i, int x, int r)
{
  if (PRT[i]>=x)
    return (r+C[i]);
  else
    return (r);
}
```

Finally, the system is initialized using the following system declarations.

SYSTEM DECLARATIONS

```
S0 = SCHEDULER(PR[0],PRT[0],C[0],D[0],0);
S1 = SCHEDULER(PR[1],PRT[1],C[1],D[1],1);
S2 = SCHEDULER(PR[2],PRT[2],C[2],D[2],2);
```

system PERIODIC_TASKS,S0,S1,S2;

If we are only interested in testing for the worst case, then all schedulers can be initialized simultaneously. However, if we are testing other properties, it may be important to initialize the system with just one scheduler S0, S1, or S2.

The global channels, $job[0]$, $job[1]$, and $job[2]$, are used by the PERIODIC_TASKS automaton to tell the SCHEDULER automaton that a new job from task τ_0 , τ_1 , or τ_2 , respectively, has been released.

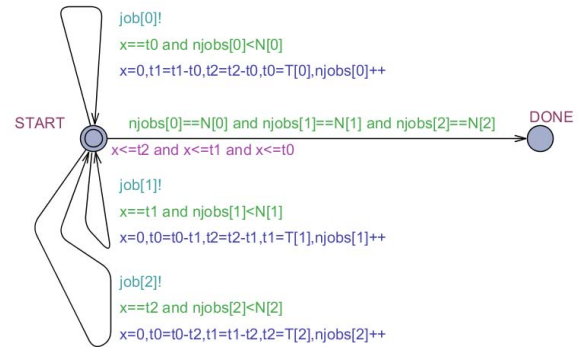


Figure 1. PERIODIC_TASKS automaton

A local clock, x , is used to determine when the jobs are released. Local integers, $t0$, $t1$, and $t2$, are used to keep track of when the next job is to be released in each task, relative to the local clock x . Since, x is set back to 0 when each job is released, the release times are also updated then as well. Finally, the number of jobs released is recorded in $njobs[i]$ for each task τ_i .

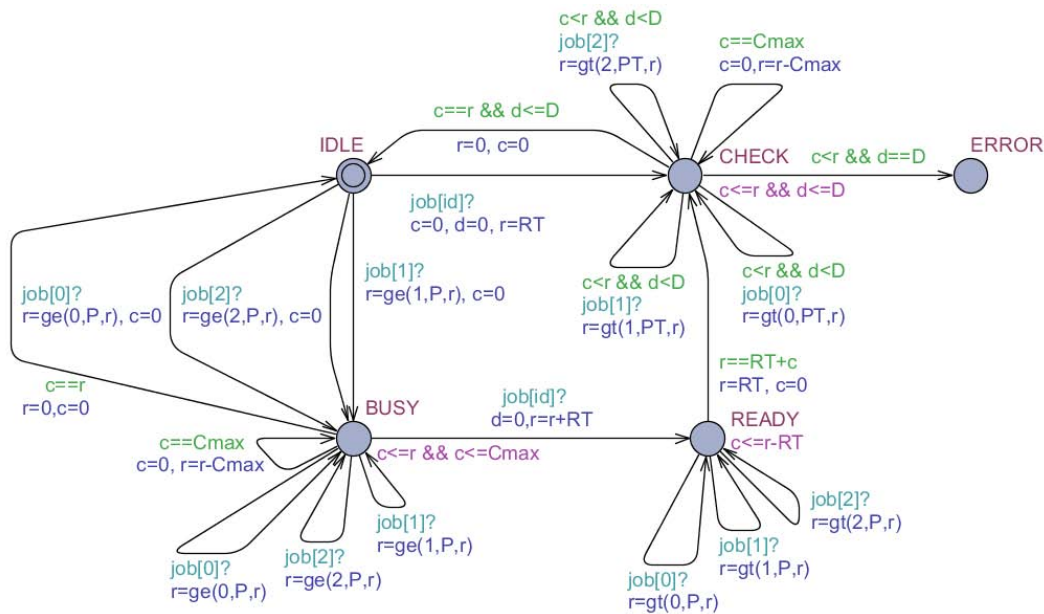


Figure 2. SCHEDULER automaton

PERIODIC_TASKS

```

clock x;
int[0, T[0]] t0;
int[0, T[1]] t1;
int[0, T[2]] t2;
int njobs[3];

```

The PERIODIC_TASKS automaton, shown in Figure 1, is essentially the same as the job generating automaton used to generate periodic jobs in previous models. The main differences are in the SCHEDULER automaton shown in Figure 2.

The intuition is the same as that used for traditional real-time response time analysis. In this model, the highest priority task is designated as task τ_0 , and the lowest priority task is designated as task τ_2 . When a message is received on channel $\text{job}[i]$, a job has been released in task τ_i . Some job in task τ_i is non-deterministically selected to be checked for feasibility. The model checker will check all possible jobs in each task. The intuition behind the model is that if some job will miss its deadline, then all jobs in task τ_i that meet their deadlines will be processed in the BUSY state. When the job that will miss its deadline is released, a transition is taken to the READY state. When the job being analyzed is scheduled for execution, the scheduler enters the CHECK state. When the deadline is reached before the job has finished executing, a transition is taken to the ERROR state. If the first job will miss its deadline, then a transition can be taken directly from the IDLE state to the CHECK

state. We rely on the model checker to test all possible jobs in each task, τ_i , to determine if there is a job that will miss its deadline.

Using the UPPAAL Verification Tool, a user can quickly check to see if the SCHEDULER ever enters the ERROR state by using the property: $E\langle\rangle(Si.ERROR)$ for $i=0,1,2$. If the property is satisfied then, on some path, the SCHEDULER automaton eventually enters the ERROR state indicating that the task set is not schedulable. For the sample data given, using a deadline monotonic priority assignment, this property is satisfied. If some job misses its deadline, a trace can be generated and visualized in UPPAAL leading to the simulator output shown in Figure 3. Note that this is the output that results when purely preemptive scheduling is used; that is, $\gamma_i = \pi_i$ for all i . Note that γ_i is denoted as $\text{PRT}[i]$ in the model.

```

const int PR[3] = { 3, 2, 1 };
const int PRT[3] = { 3, 2, 1 };

```

Recall that the priorities and preemption thresholds are specified in arrays $\text{PR}[3]$ and $\text{PRT}[3]$, respectively. To see that the SCHEDULER automaton S2 enters the ERROR state eventually on some path, we can verify that the property $E\langle\rangle(S2.ERROR)$ is satisfied, as shown below in Figure 4.

When the SCHEDULER automaton S2 goes from the CHECK state to the ERROR state, the value of $S2.c = 100$, but $S2.r = 115$ and $S2.d = 100$; that is, the first job in task τ_2 has run for 100 of 115 time units

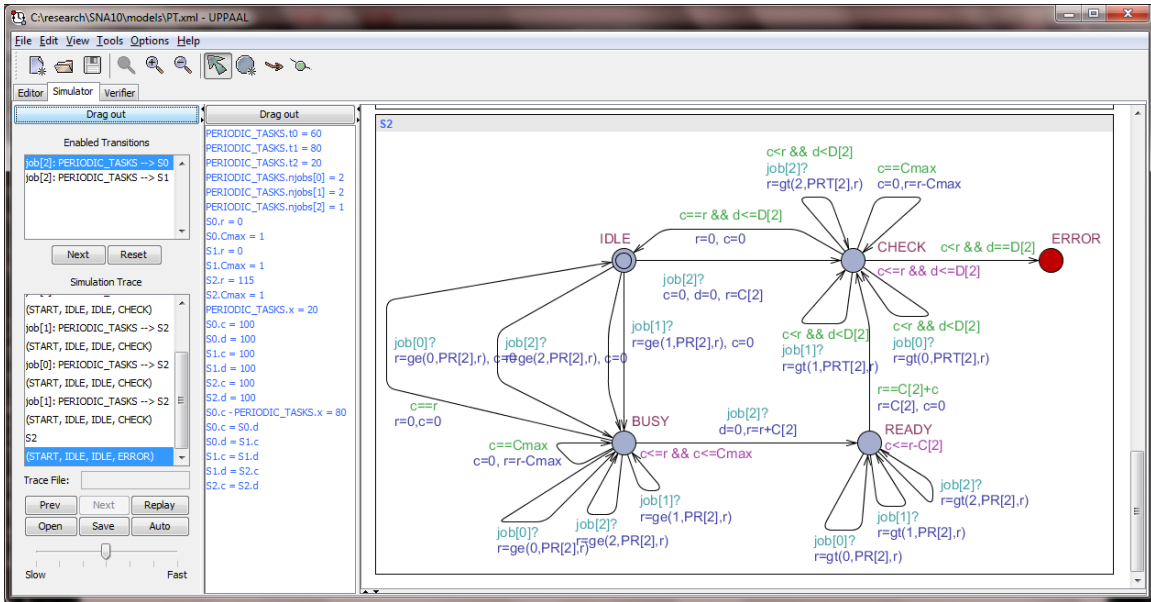


Figure 3. Missed deadline

needed to complete its execution when it misses its deadline at time 100 ($D_2 = D[2] = 100$). Even though it misses its deadline, the worst-case response time is also shown to be 115 as shown below in Figure 4.

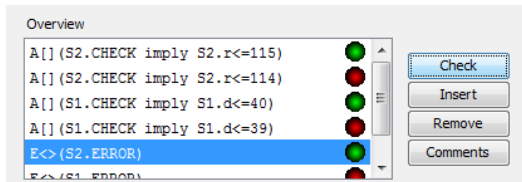


Figure 4. Preemptive tasks output

If we increase the preemption thresholds of the tasks as shown in Table 1, then the task set is schedulable. In this case, task τ_2 the low priority task, has a deadline of 100 and a period of 200. This can be easily done in the model by simply changing the constants in the global declarations; that is, changing the $PRT[3]$ array declaration to $const int PRT[3] = \{ 3, 3, 2 \}$.

Using the UPPAAL Verification Tool, it is possible to compute the worst-case response time for each task that meets its deadline using the property: $A[] (S2.CHECK \text{ imply } (S2.d \leq 95))$ which is satisfied. The same property with an upper bound of 94 is not satisfied, as shown in Figure 5. Thus, the worst-case response time of task τ_2 is 95, which is less than its deadline of 100.

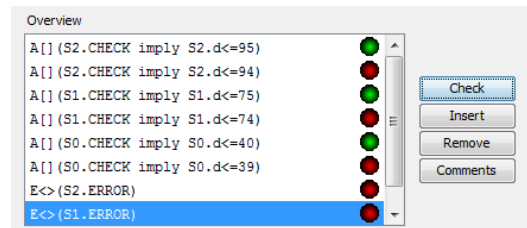


Figure 5. Verification output

In addition to scheduling properties, it is possible to analyze other safety and liveness properties using the model checker UPPAAL.

As noted in [6], the value used for C_{max} to decrement r can be any positive value. It turns out that if C_{max} is larger than the level- i busy period, then the number of states generated and searched will be the least possible, but the value of r may reach the length of the level- i busy period.

The number of states searched for each value of C_{max} is shown below in Table 2. This is based on the case when the preemption thresholds are set to 3, 3, and 2, respectively.

Table 2. Bounds on number of states.

C_{max}	States	C_{max}	States
1	150,180	50	1,793
3	20,820	100	1,635
6	8,155	250	1,617
12	3,805	500	1,617

If we remove the transitions involving C_{max} , then the number of states searched and stored is 1,617. The

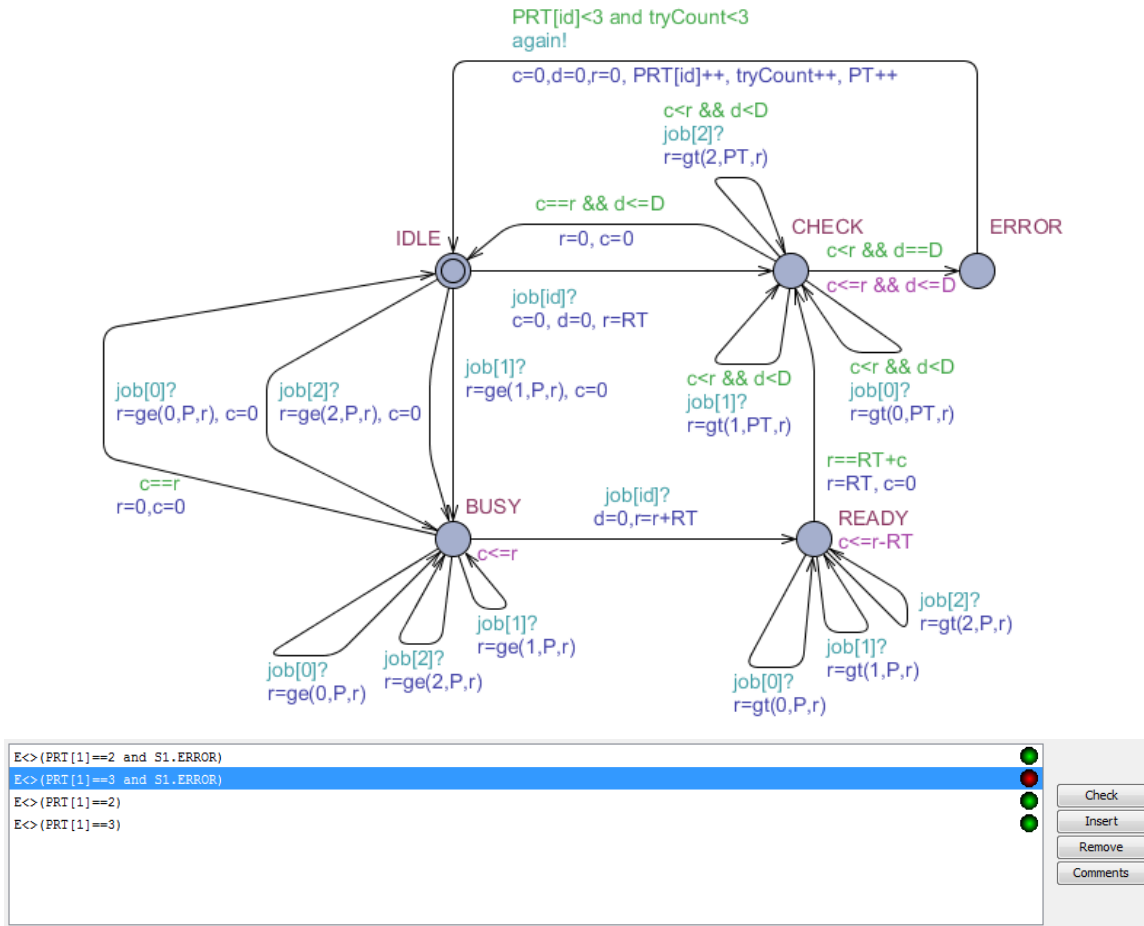


Figure 6. Updated SCHEDULER automaton

number of states is directly related to the amount of time required to evaluate the property.

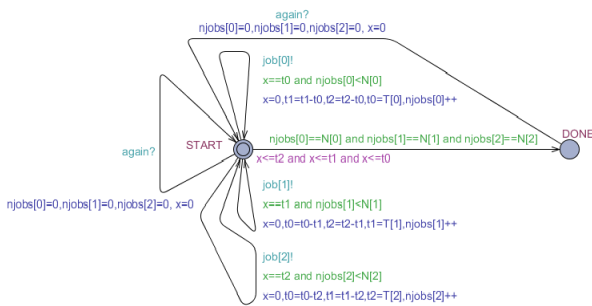


Figure 7. Updated PERIODIC_TASKS automaton

Finally, if we set the preemption thresholds to be as large as possible, then non-preemptive scheduling is realized; e.g., set `const int PRT[3] = 3,3,3`. UPPAAL queries can be used to verify that the highest priority task misses its deadline.

The scheduler and job creation automata can be modified to find an optimal assignment of preemption thresholds that are minimal (preemption thresholds as small as possible). To find an optimal assignment that is minimal, the SCHEDULER automaton shown in Figure 6 can be used. If the current assignment leads to an error, then the preemption threshold is incremented, and the automaton is reset by sending a message on the *again* channel to try again as long as the preemption threshold is less than the maximum priority. The SCHEDULER automaton goes from the ERROR to the IDLE state, and the PERIODIC_TASKS automaton goes back to the START state, and starts sending the same set of jobs all over again.

For the second task in our example, the threshold must be elevated to the maximum priority 3 to ensure that the task set is feasible. As shown in Figure 6, if the preemption threshold of task 1 is initialized to 2, then the ERROR state is reachable, but if the preemption threshold is initialized as 3, then the ERROR state is not reachable. Similarly, the SCHEDULER

automaton could be adapted to compute the maximum feasible preemption threshold, in most cases we are only interested in finding the smallest possible preemption threshold that will work.

4 Conclusions

This paper presented a simple model that can be used to analyze the feasibility of periodic task sets with preemption thresholds and compute an optimal assignment of preemption thresholds. The model consists of a set of two automata: a PERIODIC_TASKS automaton to model the release of periodic jobs, and a SCHEDULER automaton to model the scheduler. The model generated can be used with UPPAAL 4.0 to determine if the task set is feasible, and to compute the worst-case response times of tasks that meet their deadlines. If the task set is not feasible, the model can also be used to identify why the task set fails to meet its deadlines. The example UPPAAL models and queries generated are available on-line at: <http://www.cis.ksu.edu/~neilsen/PDPTA11/>.

The model and queries could easily be generalized to test the feasibility of other periodic task sets with arbitrary start times and arbitrary deadlines. UPPAAL is a very powerful tool that can be used for many types of verification for distributed and real-time systems. An interesting next step will be to incorporate dynamic voltage scheduling into the model and verify some of the properties specified in [8].

References

- [1] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *In Proc. of FORMATS03, number 2791 in LNCS*, pages 60–72. Springer-Verlag, 2003.
- [2] G. Behrmann, A. David, K.G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of SysTems (QEST) 2006*, IEEE Computer Society, pages 125–126, 2006.
- [3] J. Bengtsson, F. Larsson, P. Pettersson, W. Yi, P. Christensen, J. Jensen, P. Jensen, K. Larsen, and T. Sorensen. Uppaal: a tool suite for validation and verification of real-time systems, 1996.
- [4] J. Chen, A. Harji, and P. Buhr. Solution space for fixed-priority with preemption threshold. In *11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS 05)*, pages 385–394, 2005.
- [5] E. Fersman, P. Pettersson, and W. Yi. Timed automata with asynchronous processes: schedulability and decidability. In *In Proceedings of TACAS 2002*, pages 67–82. Springer-Verlag, 2002.
- [6] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis using two clocks. In *In 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, pages 224–239. Springer, 2003.
- [7] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science*, 354(2):301 – 317, 2006. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003).
- [8] R. Jejurikar and R. Gupta. Integrating preemption threshold scheduling and dynamic voltage scaling for energy efficient real-time systems. In *International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 04)*, 2004.
- [9] M.L. Neilsen. A generalized token-based mutual exclusion algorithm for wireless networks. In *Proceedings of the 20th Int'l Conference on Parallel and Distributed Computing Systems (PDCS-2007), Las Vegas, Nevada, USA*, ISCA, 2007.
- [10] M.L. Neilsen. Model checking token-based distributed mutual exclusion algorithms. In *Proceedings of the 15th International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 10–16, 2009.
- [11] M.L. Neilsen. Symbolic schedulability analysis of task sets with arbitrary deadlines. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'10)*, 2010.
- [12] M.L. Neilsen. Symbolic schedulability analysis of task sets with preemption thresholds. In *Second International Sensor Networks and Applications Conference (SNA'10)*, 2010.
- [13] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *RTCSA*, pages 328–, 1999.

Analysis of False Cache Line Sharing Effects on Multicore CPUs

Suntorn Sae-eung, M.S.

suntorn.saeung@students.sjsu.edu

(408) 775-9883

Robert Chun, Ph.D.

Computer Science Department

San Jose State University

San Jose, CA, 95192-0249 USA

robert.chun@sjsu.edu

(408) 924-5137

ABSTRACT

False Sharing (FS) is a notorious problem occurring in multiprocessor systems. It results in a performance degradation for multi-threaded programs. Since the architecture of a multicore processor is very similar to that of a multiprocessor system, the presence of the false sharing problem is speculated. Its effects should be measurable in terms of efficiency degradation in a concurrent environment on multicore systems.

This article discusses the causes of the false cache line sharing problem in dual-core CPUs, and demonstrates how it lessens the system performance by measuring speed-ups and efficiency of the experiments in sequential compared to parallel executions. Thus, demonstration programs are developed to collect the execution results of the test program with and without false sharing on the specific system hardware. Certain techniques are implemented to eliminate false sharing. These techniques are described, and their effectiveness in mitigating the speed-up and efficiency lost from false sharing is analyzed.

KEY WORDS

False Sharing, Cache Memory, Spacing, Padding

1.0 Introduction

The current trend of processor design is towards multicore CPUs. Recently, eight-core and twelve-core CPUs have been in the manufacturing process for both AMD and Intel [1]. Processor manufacturers overcome the heat-wall constraint by packing more than one computing module, so-called cores, into a package. Sometimes the chip is simply referred to as a Chip Multiprocessor (CMP); however, a processor can also be coined by the number of its cores. For example, a two core processor is called as a “dual core” CPU. Having many processing cores working together increases complexity in hardware design and software production. The hardware manufacturer is not the only party involved in taking advantage of the multiple core processors. Programmers must also understand how to make use of additional cores, and design the application by dividing processes into several sub-tasks, and assign them to several threads to utilize all available computing cores.

A potential problem in multiprocessor systems that can cause poor performance by mistakenly updating data in a shared cache line is the “false sharing” problem. Previous research on multiprocessor systems demonstrated huge impacts of the false sharing problem [5][6][8][9][12]. It can cause performance degradations of 20x on a four-processor system, and 100x on an 8-processor system.

Because multiprocessor and multicore architectures are similar, we hypothesize that FS can occur on multicore systems too. This paper demonstrates the existence of false sharing on systems with dual core CPUs, measures the impact of the false sharing issue, and compares the performance drops caused by false sharing between a dual core processor to a multiprocessor system.

1.1. Memory hierarchy and cache elements

Levels and types of memories are distinguished by their access time, capacities and complexities. Certain types of CPUs, along with their cache and main memory are selected as representatives to illustrate the memory hierarchy of multiprocessor and multicore systems. As false sharing is previously notorious in multiprocessor systems, the memory architecture of a Symmetric Multiprocessor (SMP) is compared with that of a Chip Multiprocessor (CMP).

1.1.1 Memory architecture in Symmetric Multiprocessor

The Symmetric Multiprocessor (SMP) is a classical configuration for a multiprocessor system. The memory hierarchy of SMP is categorized in two levels: cache memory and main memory. CPU access time, or latency, on the cache is far less than that from the main memory. Processors use the cache memory as a local memory, and consider the main memory to be a remote memory. CPUs need to request data through a shared network, bus, or crossbar in order to read from and write to the main memory. A simple diagram of a SMP system is shown in figure 1.

1.1.2 Memory architecture in Chip Multiprocessor

Chip Multiprocessor (CMP) is a way to name multicore processors. The cache in a CMP system is divided into tiers similar to a SMP, but a CMP’s structure adds more layers of caches, e.g. a cache level 2, interleaving the L1 cache and the main memory so as to reduce the latency gap between the upper and the lower memory layers as shown in figure 2.

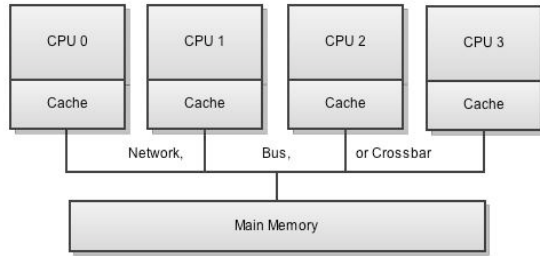


Figure 1. Memory hierarchy in SMP [4]

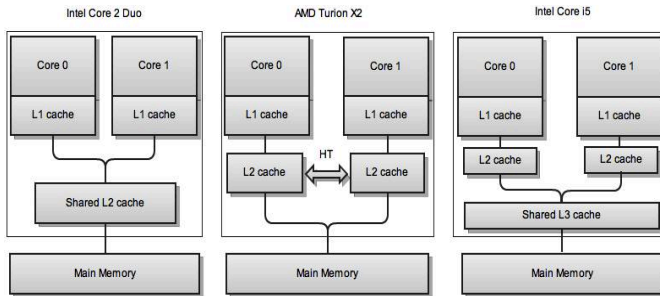


Figure 2. Memory hierarchy in CMP

The diagram shows three distinct layouts of caches. The Intel processor (left) implements a shared L2 and L3 cache enabling all cores to access shared data. AMD CPUs (middle) make use of a special dedicated hardware and protocol, Hyper Transport technology, to synchronize shared data between each core's L2 cache. Recently, a more advanced CPU, such as the Intel Core i5, a processor is composed of two levels of separate caches, and a shared L3 cache (right).

1.1.3 Cache lines

A cache line is the smallest unit of data that can be transferred between the main memory and the cache. The cache line size varies by processor makers; the size can be directly obtained from the processor's specification sheets, or retrieved by executing some manufacturer-provided instruction sets. All processors in our test hardware have a 64-byte cache line size.

1.2 Multiprocessor/multicore cache coherency

For a multiprocessor system, all processors typically have their own caches, and machine vendors must ensure that data across processors are coherent. A protocol must be used to enforce data consistency among all the cores' caches so that the system correctly processes valid data; this protocol is called a "cache coherency" protocol. The protocol manages data to be updated appropriately using a write-back policy, resulting in decent overall performance by reducing the number of main memory updates.

1.3 False cache line sharing

False cache line sharing or *false sharing* in short is a form of cache trashing caused by a mismatch between the memory layout of write-shared data across processors and the reference pattern to the data. It occurs when two or more threads in parallel programs are assigned to work with different data elements in the same cache line. In other words, *false sharing* is a side effect in a multiprocessor system due to cache coherency.

Although the multiprocessor's system scale seems quite different from that of a small personal computer, the internal architecture of a multiprocessor is comparable to a multicore microprocessor chip in terms of the number of processors and memory hierarchy. A computer with dual-core, quad-core, or octal-core processors is now considered as a type of multiprocessor system. Thus, it would seem to be susceptible to a *false sharing* problem as well.

A multiprocessor system must maintain data coherency across CPUs. When a processor makes a change on its cache, other processors must be aware of the change, and determine whether its copy of data in cache needs to be reloaded or not. The cache coherency protocol defines rules to maintain data updates among processor groups with a minimal number of requests to the main memory, thereby optimizing system performance.

False sharing occurs when threads from different processors modify variables which reside on the same cache line. In case of Intel's processors, when the processor invalidates a cache line with an outdated value, it fetches an updated value from the main memory into its cache line to maintain data validity. Figure 4 and 5 demonstrate two threads with *false sharing* on SMP and CMP systems respectively. Threads 0 and 1 update variables that are adjacent to each other located on the same cache line. Although each thread modifies different variables, the cache line keeps being invalidated every iteration. As a result, the number of the main memory access increases considerably, and causes great delays due to the high latency in data transfers between levels of the memory hierarchy. The *false sharing* problem is occasionally referred to as "cache line ping-pong [9]."

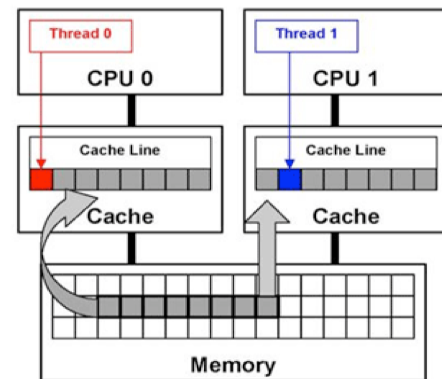


Figure 4. False cache line sharing on SMP [5]

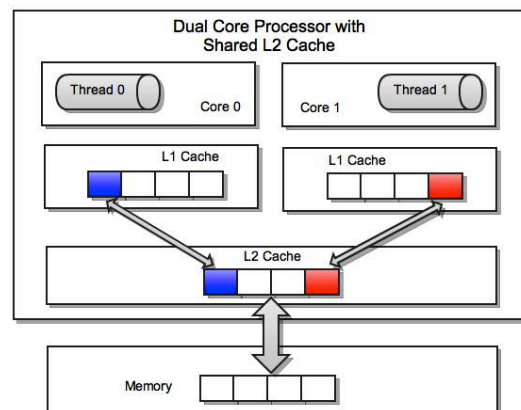


Figure 5. False cache line sharing on CMP [5]

2.0 Prior Work

Many researchers point out the great performance degradation caused by the *false sharing* problem on multiprocessor environments. Fewer papers performed tests on multicore CPUs since they are a relatively new architecture. The hypothesis in this paper is that *false sharing* would happen in a multicore system as it does in a multiprocessor one because it has many common components, yet the degree of impact may be different. More details will be discussed in the experiment and the result section.

2.1 Concurrent Hazards: False sharing

Butler did an experiment on a multiprocessor system to measure *false sharing* effects in [6]. The test system is composed of four dual core CPUs, total 8 processing units. The best case speed-up at the eight-threaded execution shows a 100x difference compared to the worst case. The paper employed certain techniques to eliminate *false sharing* effects.

2.2 Latency of conflict writes on Multicore Architecture

Dr. Josef discussed the latency penalty caused by FS [7]. The work shows that the amount of latency declines when the array allocated is between 128 Kbytes and 2Mbytes in size, which fits on cache level two. At this threshold of the array size, the high latency that would have been caused by the *false sharing* problem disappears. It is because shared L2 cache is a “true” sharing cache, and both cores can access data without cache invalidation, thereby eliminating *false sharing*. In brief, the experiment proved that shared cache between cores can eliminate the adverse impact stemming from *false sharing*.

3.0 Experiment Design

The experiment results are obtained from the execution time of a designated program onto three systems with different types of dual core CPUs. There are five test cases which have the same goal of completing an equally specified workload; however, different running schemes are set up to reveal the existence of the *false sharing* problem.

3.1 False sharing avoidance techniques

Since *false sharing* results from two or more cores using data in the same cache line, one way to get rid of it is to eliminate any sharing in the same cache line. Hence, certain techniques are proposed in order to avoid data sharing by modifying the data arrangement in the cache line.

3.1.1 Spacing technique

The Spacing technique is an approach used to split a contiguous allocated space. In an array, a set of variables is typically reserved in a chunk to take advantage of locality of reference. For instance, when four variables are declared in an array, an allocation consisting of four integer-sized adjoining memory blocks is made. Using the Spacing technique splits the shared data among the reserved array by shifting the offset between each contiguous array element so that each element resides on a separate, different cache line.

In figure 6a, integers D1, D2, D3 and D4 reside in the same cache line. With the implementation of the Spacing technique, *false sharing* on array data can be avoided as shown in figure 6b.

3.1.2 Padding technique

Besides the Spacing technique, Padding is another technique to reduce *false sharing* effects by filling a cache line with a pad.

A variable declaration requires an extra piece of information to manage memory space for the variable. When an array is declared, the operating system needs to define metadata that contains the array information. Metadata uses space just right before actual data, and consists of pointers and header information. For example, every array in .NET require metadata such as SZARRAY, which stores size information of the array. Whenever a thread read from or writes to an element, there is a read of the metadata happening before that of the actual array. Using Spacing technique does not separate metadata from the array; they still reside on the same cache line as in figure 6b. Therefore, *false sharing* is happening between the metadata and the first array element. To eliminate sharing on metadata, the cache line is padded so that the first element is shifted to the next cache line. Figure 6c illustrates the cache line structure using Padding.

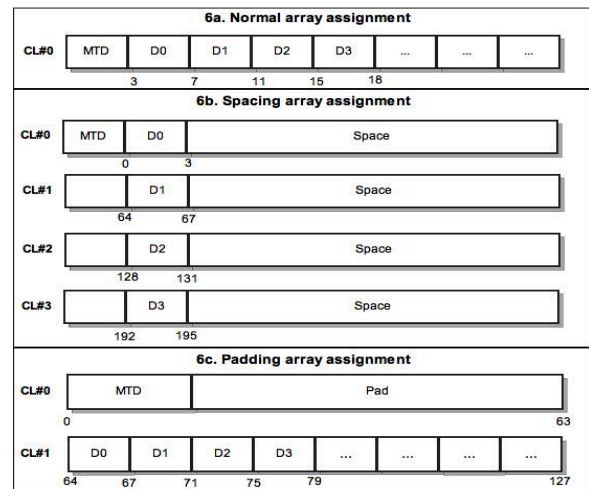


Figure 6. Cache line structure of Spacing, and Padding arrays

3.1.3. Combined Spacing and Padding technique

Using a Spacing-only or a Padding-only technique would not overcome the *false sharing* problem [6]. Therefore, the combination of both techniques is the best way to completely avoid *false sharing* by isolating each elements onto a single cache line. Figure 7, for example, shows a cache line layout of four array elements, including metadata.

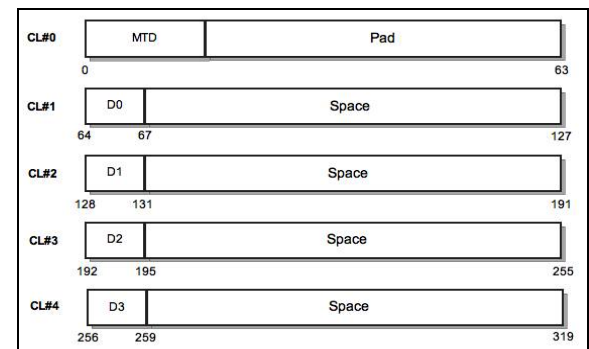


Figure 7. Cache line structure of combined Padding and Spacing technique

3.2 Testing code

The testing code, which is adapted from [6], demonstrates existence of the *false sharing* problem. The processing time of the program with the *false sharing* problem is compared to the program without the problem. The identical experiment is executed on three hardware configurations to compare the performance loss among different systems. The size of the Padding and Spacing variables are defined to be 64 bytes, which is equal to a size of one cache line, to ensure that every element is shifted off onto a separate cache line. The data arrangement is the crucial focus in order to avoid *false sharing* with five testing cases: SEQ, PAR, PAR_SPC, PAR_PAD and PAR_SPC_PAD.

The following code fragments show how each testing case declares the data array, sets an offset, and executes the workload.

```
var data = new int[ _Padding + ( _ThreadCount * _Spacing ) ];
...
var offset = _Padding + ( iThread * _Spacing );
...
for ( int x = 0 ; x < iters ; x++ ) data[ offset ]++;
```

For example, suppose that a system consists of a four core processor working on four integer elements; each core works on an array element. In the Parallel FS case (PAR), all four threads work on the contiguous array elements as shown in figure 11a. The array data is arbitrarily defined to start at the memory address 156. Generally an integer requires four bytes of memory space; therefore, four integers can be allocated in one cache line. Figure 8a show cache diagram of PAR case that *false sharing* occurs on cache lines. Meanwhile, figure 8b illustrates how PAR_SPC_PAD case avoids *false sharing* effects by isolating each array element onto separate cache lines.

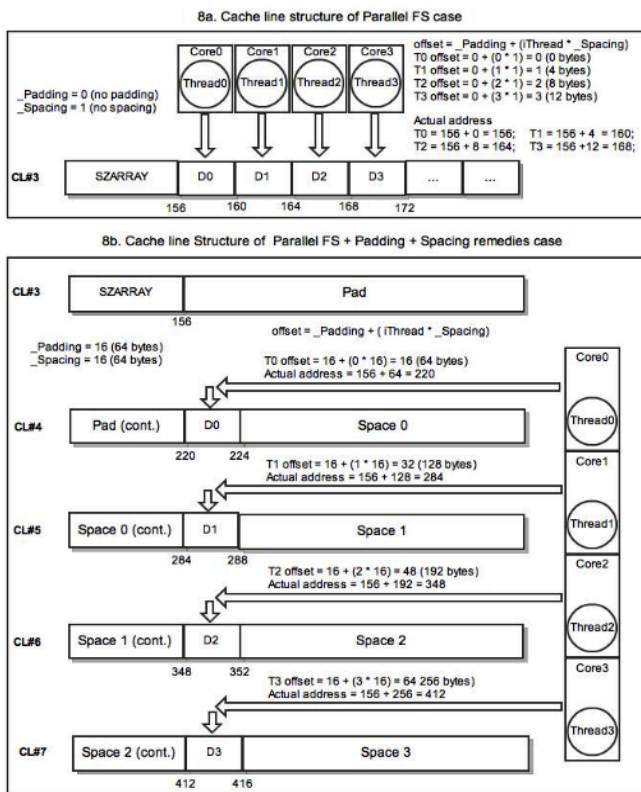


Figure 8. Cache line structures of Parallel FS (PAR) and Parallel FS + Spacing and Padding remedies (PAR_SPC_PAD)

4.0 Hardware specifications

The experiment performs on three specified types of multicore processors: Intel Core2 Duo T5270, AMD Turion64 X2 TL-58, and Intel Core i5 520M.

Table 1. Processors' specifications of testing systems

Hardware	CPU	L1 Cache	L2 Cache	L3 Cache
Dell Vostro 1400	Intel Core2 Duo T5270 1.4GHz	32kbytes/core (Inst and Data)	Shared 2Mbytes	-
HP DV6000	AMD Turion64 X2 TL-58 1.9GHz	64kbytes/core (Inst and Data)	512kbytes/core (Unified)	-
Macbook Pro MC371LL/A	Intel Core i5 520M 2.4GHz	32kbytes/core (Inst and Data)	256kbytes/core (Unified)	Shared 3Mbytes

5.0 Experiment Results

The experiments results are collect and analyzed to understand how *false sharing* happens, and how much performance degradation it causes. The runtime values of five different test cases with varied data layouts and running schemes are collected. All five cases are assigned to complete the same amount of workload so that they can be compared in terms of performance. The details of data arrangement in each case are as follows.

1. Sequential (SEQ)—a sequential execution of the assigned workload on one core.
2. Parallel FS (PAR)—an execution of the assigned workload on all available cores in parallel. The amount of workload is divided equally for every core. There will be data contention in cache lines. The runtime on this case is expected to be influenced by *false sharing*.
3. Parallel FS + Spacing remedy (PAR_SPC)—an execution of the assigned workload on all available cores in parallel. The amount of workload is divided equally for every core. Additionally, this case applies the Spacing technique to avoid *false sharing* effects on the array elements.
4. Parallel FS + Padding remedy (PAR_PAD)—an execution of the assigned workload on all available cores in parallel. The amount of workload is divided equally for every core. This case implements the Padding technique to prevent *false sharing* occurring on the array metadata.
5. Parallel FS + Spacing and Padding remedies (PAR_SPC_PAD)—an execution of the assigned workload on all available cores in parallel. The amount of workload is divided equally for every core. Moreover, this case combines Spacing and Padding techniques so as to completely eliminate *false sharing* effects on the array elements and metadata.

5.1 Experiment results

Speed-up and efficiency are calculated from the runtime. Both numbers are computed as relative parallel performance based upon the sequential runtime by following equations [3].

$$\text{Speed-up}(x) = \text{sequential runtime} / \text{parallel runtime} \quad (1)$$

$$\text{Efficiency} (\%) = (\text{speed-up} / \text{number of cores}) * 100 \quad (2)$$

Table 2 amasses all experiment results which are speed-ups, efficiency as well as performance degradation computed in terms of loss efficiency. Values of each hardware configurations are relatively compared based upon sequential execution figures.

Table 2. Experiment results summary

System \ Test cases	SEQ	PAR	PAR_SPC	PAR_PAD	PAR_SPC_PAD
Intel Core2 Duo T5270					
Speed-ups (x)	1.00	0.51	0.76	0.99	1.75
Efficiency (%)	100	25.48	37.85	49.43	87.58
Loss (%)	-	74.52	62.15	50.57	12.48
AMD Turion 64 X2					
Speed-ups (x)	1.00	0.50	0.73	0.63	1.97
Efficiency (%)	100	25.12	36.28	31.3	98.34
Loss (%)	-	74.88	63.72	68.7	1.66
Intel Core i5 520M					
Speed-ups (x)	1.00	0.57	1.06	0.98	2.17
Efficiency (%)	100	28.48	52.89	49.21	108.57
Loss (%)	-	71.52	47.11	50.79	0 (+8.57)

Intel Core2 Duo T5270

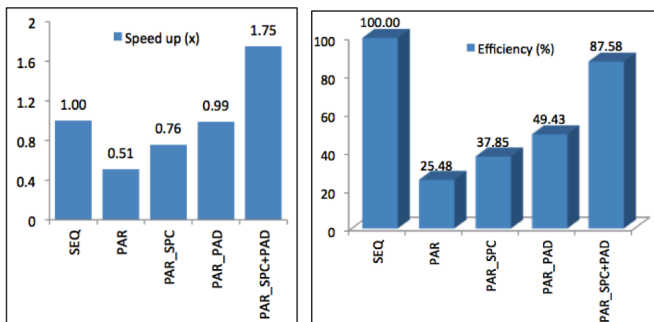


Figure 9. Intel Core2 Duo T5270 speed-ups and efficiency

AMD Turion64 X2 TL-58

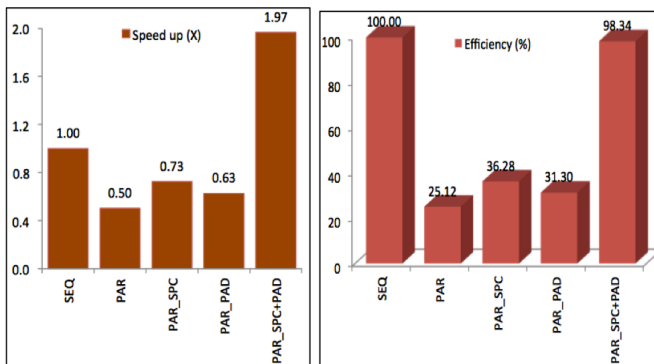


Figure 10. AMD Turion64 X2 TL-58 speed-ups and efficiency

Intel Core i5 520M

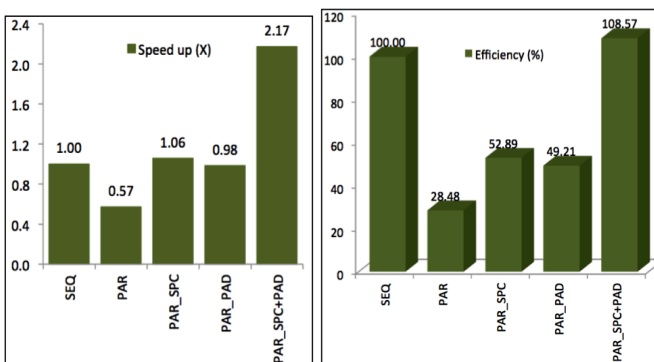


Figure 11. Intel Core i5 520M speed-ups and efficiency

5.1.1 Intel Core2 Duo T5270

Figure 9 (left) show speed-up ratios of the four parallel cases calculated based upon the Sequential case (SEQ) speed-up (1.0x).

The speed-up ratios demonstrate that *false sharing* has the most influences on PAR case execution (0.51x), and less impacts on the two cases with remedial techniques, PAR_SPC (0.76x) and PAR_PAD (0.99x). The PAR_SPC_PAD case obtains a practical value at 1.75x in speed.

Theoretically, two cores should accelerate system performance for two times (2x). However, the speed-up ratio in practical does not reach the theoretical value since some system resources are used to fork working threads, and synchronize data among those threads. A speed-up ratio range of 1.5x to 1.9x is considered practical in the level of parallelism with two processing cores [2].

Efficiency is a fairly good indicator to measure performance per processing unit. The Sequential case is a base value with 100% efficiency. For two cores working in parallel, the system must run two times faster than single core to achieve full efficiency. Figure 9 (right) shows the efficiency with a similar pattern to speed-up ratios, PAR at 25.48%, PAR_SPC at 37.85%, PAR_PAD 49.43%, and PAR_SPC_PAD at 87.58%. The amount of lost efficiency results from the different degrees of *false sharing* impact. The more *false* cache line sharing occurs, the lower performance it obtains.

5.1.2 AMD Turion64 X2 TL-58

Consider the speed-up of the PAR case, it does not scale well (0.5x) compared to the sequential case (1.0x). When the PAR case is employed with the Spacing technique to become the PAR_SPC, the speed-up augments to be 0.73x. The PAR_PAD also produces a greater speed-up (0.63x) compared to the PAR case as shown in figure 10 (left).

False sharing turns down speed-ups of the three mentioned cases in different degrees. However, the Parallel FS + Spacing and Padding remedies case (PAR_SPC_PAD) gains a promising speed-up at 1.97x, which is virtually close to an ideal value at 2.0x.

Among all parallel cases, only the PAD_SPC_PAD gains high efficiency at 98.34% as shown in figure 10 (right). The efficiency in any other cases reflects the different performance degradation by different degrees of *false sharing* effects.

5.1.3 Intel Core i5 520M

Figure 11 (left) shows speed-up ratios on the Intel Core i5 520M test system. The Parallel FS (PAR) case represents the poor performance execution with 0.57x in speed, or around two times slower than the sequential case. An improvement takes place on the PAR_SPC case (0.98x) and the PAR_PAD case (1.06x). The PAR_SPC_PAD case gains the highest speed-up at 2.17x.

Intel Core i5's efficiency has a similar pattern to two previous test systems. The efficiency of the PAR_SPC_PAD is noticeable with a "superlinear" number (108.57%), which efficiency exceeds 100% [3]. When a program which makes use of data stored in a share cache is repeatedly executed, its performance will substantially boost up because of memory locality. Another factor to reach a superlinear value is capability of executing many concurrent threads. Intel Core i5 520M processor comes up with Hyper-Threading technology; each core can execute two threads at a time. Therefore, it increases probability for threads to take advantage of memory locality.

5.2 Performance drops caused by false sharing

From table 1, performance drops caused by *false sharing* is observed by efficiency losses.

The PAR case suffers from *false sharing* the most. The system performance drops by three fourth of the speculated efficiency, which caused efficiency loss 70-75%. The PAR_SPC case and the PAR_PAD case also have significant performance degradation approximate 50-70%, but less deficit compared to the PAR case. The PAR_SPC_PAD case performs efficiently, especially on the Core i5 520M processor. The case has a small number of losses on all three systems: Intel Core2 Duo T5270 at 12.48%, AMD Turion 64 X2 at 1.66%, and no loss for Intel Core i5 520M (8.57% in surplus).

5.3 False sharing impacts comparison on multiprocessor and dual core systems

The previous research points out the severity of the *false sharing* impact on multiprocessor systems in two orders of magnitudes (-100x) [6]. However, the experiment results in this paper demonstrate the worst case of performance degradation by a factor of four (-4x). An important observation is the degree of impact on a multiprocessor system is far aggressive than that on a dual core system. The suspicious factor is memory hierarchy.

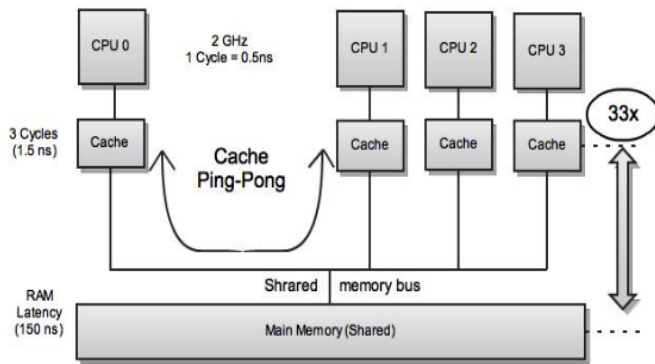


Figure 12. Cache Ping-ponging on multi-level memory in a multiprocessor system

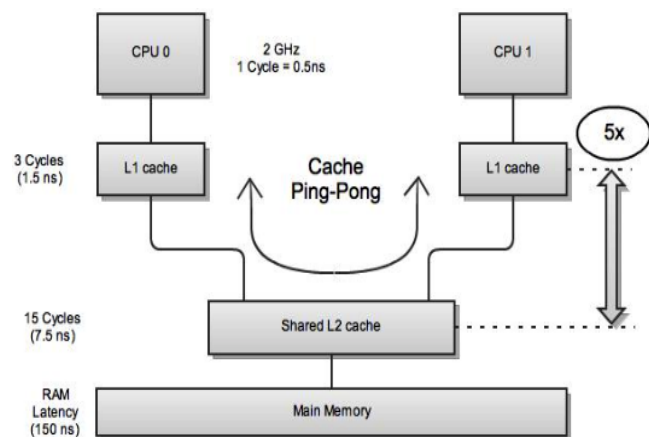


Figure 13. Cache Ping-ponging on multi-level memory in a dual core system

Figure 12 and 13 show multi-level memory hierarchies of a multiprocessor system and an Intel dual core processor system. Supposed that the program similar to the one that runs in the test experiment is executed in a multiprocessor system, *false sharing* occurs on the system. In the PAR case, the array elements in a cache line are updated by many processors; *false sharing* results in considerably numbers of cache line invalidation. When a processor writes a new value to its array elements, the whole cache line needs to be written back to the main memory, and reload to all processors' caches, known as cache Ping-Pong in figure 12. The CPUs' read and write operations befall between their caches and the (shared) main memory, in other words, between the cache and the main memory hierarchy. Since the processors need to access to the main memory through a shared bus, the system suffers from cache misses penalty. The amount of CPU waiting time substantially increases by the cache miss penalty as a following equation [10]:

$$\text{Cache miss penalty (X bytes)} = \text{main memory access latency} + X \text{ bytes/data receive rate} \quad (3)$$

Cache miss penalty is computed by adding up a delay of main memory access and data transfer time from main memory to cache memory. The data transfer rate depends on the shared memory bus. Because the bus is used by all processors to access to main memory and peripheral devices, transfer time of the bus has much higher latency than that of an internal bus between caches and CPUs. Therefore, the substantial amount of increasing time caused by cache miss penalty results in significant performance reduction stemmed from the *false sharing* problem.

The similar scenario of *false sharing* occurs on a dual core system. Cache Ping-Ponging also happens in the system as shown in figure 13. Yet, the cache invalidation in the dual core system takes place in between the L1 cache and the shared L2 cache, instead in between the cache and the main memory in multiprocessor systems. The on-die caches are local memories having low latency. Data transfers among caches do not require bus transactions as data transfers between cache and main memory. Thus, the severity of *false sharing* on a dual core system does not cause significant performance degradation as it does on a multiprocessor system.

6.0 Conclusion

The study of *false sharing* effects on dual-core CPUs demonstrates the existence of *false sharing* on multicore CPUs. The issue apparently degrades overall performance in a concurrent execution.

- (1) In the test case with false sharing occurs, PAR, on dual core processors, the efficiency degrades by approximately 70-75%. In other words, the test program works slower than speculated by four times; it runs at 25-30% efficiency instead of 100% efficiency.
- (2) For the partially *false sharing* remedial cases, PAR_SPC and PAR_PAD, have certain runtime improvements to be 30-50% efficiency. However, the *false sharing* impact still stalls the two test cases, and leads to significant efficiency loss.
- (3) On the best case, PAR_SPC_PAD, completely avoids *false sharing*, and obtains performance at nearly 100% efficiency.

All three test systems, Intel Core2 Duo T5270, AMD Turion64 X2, and Intel Core i5 520M processors, are consistently suffering from *false sharing* effects resulting in performance drops at 50%-75% efficiency.

On one hand, programmers can be optimistic for improvements on multicore CPUs since the ratio of performance drops caused by the *false sharing* problem on a dual core system is not as high as that on a multiprocessor system. The findings in this paper indicates performance of a dual core system drops approximately by a factor of four (-4x). Unlike the *false sharing* impact on a multiprocessor system, the previous research reported the performance loss as high numbers as one hundred times (-100x) on an eight processor system. The different degrees of the *false sharing* impacts stem from the different memory architectures between those two systems. The shared cache implementation on Intel dual core processors alleviates the adverse impact caused by *false sharing*. For AMD processors, although each core has a separate L2 cache which is subject to have *false sharing* problems, the processor handles the data synchronization among caches on all cores by using MOESI coherency protocol and dedicated data paths, the synergy of the two parts are named as Hyper Transport technology. In brief, both Intel and AMD have deliberately come up with the intelligent designs to cope with the data sharing issue across cores.

On the other hand, the programmers must still be aware of performance degradation caused by *false sharing*. For dual core processor system, a parallel version of the program working four times slower is considered unacceptable since it runs even slower than sequential version running on a single core processor. The *false sharing* problem, therefore, is a major potential issue in parallel programming on multicore CPUs.

We proposed and demonstrated implementation of Spacing and Padding techniques to avoid *false sharing*. These approaches remedy, or totally eliminate, the *false sharing* impact. Nevertheless, the implementation of Spacing and Padding techniques barter with memory space. For instance, on the dual core test systems, the amount of memory used in the PAR case is 8 bytes of the array plus the metadata size, which can be rounded up to be 16 bytes. The modified array size in the PAR_SPC_PAD case becomes three cache lines, or 192 bytes, two cache lines for two elements and one cache line for metadata. Thus, the cost to avoid *false sharing* is rather expensive.

7.0 Future Work

The processors with four cores, six cores, and eight cores will be a standard for personal computers in the foreseeable future. Also, the internal architecture of processors keeps changing to handle inter-core communication efficiently. For Intel Core-i7, data on each core is synchronized through inter-core connection paths known as Intel Quick Path technology [1]. AMD Phenom X4 Quad-core uses Hyper Transport 3.0 technology maximizing throughput to be 51.2Gbit/second [11]. All break-through technologies are invented to tackle data synchronization among cores. However, does the new cutting edge technology really work on all types of applications without the *false sharing* issue? If it does, that is good news for programmers. This paper shows the existence of *false sharing* on dual core CPUs, and it could imply that false sharing would still occur on a more-than-two-core processor. In case the problem does exist, how much is the impact on a quad core CPU? How much is the performance loss on an eight core or a sixteen core processor? The evaluation of the *false sharing* impact on such many cores CPUs will be subject to further research in the future.

8. References

- [1] AMD, Intel ready 'many core' processors. Web site: http://news.cnet.com/8301-13924_3-10471333-64.html
- [2] Pase, M. D., Eckl, M.A. 2005. A Comparison of Single-Core and Dual-Core Opteron Processor Performance for HPC. IBM Corporation. Web site: ftp://ftp.software.ibm.com/eserver/benchmarks/wp_Dual_Core_072505.pdf
- [3] The Code Project. Butler, N. Superlinear: an investigation into concurrent speed-up. Web site: <http://www.codeproject.com/KB/threads/Superlinear.aspx>
- [4] Loshin, D., Effective Memory Programming. McGraw-Hill.
- [5] Chandler, D., Reduce False Sharing in .NET. Web site: <http://software.intel.com/en-us/articles/reduce-false-sharing-in-net/>
- [6] The Code Project. Butler, N. Concurrent Hazards: False Sharing. Web site: <http://www.codeproject.com/KB/threads/FalseSharing.aspx>
- [7] Weidendorfer, J., et al. 2007. Latencies of Conflicting Writes on Contemporary Multicore Architectures. *Springer Berlin Heidelberg*, vol. 4617, pp. 318-327.
- [8] Bolosky, W. J., Scott, M. L. 1993. False sharing and its effect on shared memory performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4 (Sedms'93)*, Vol. 4. USENIX Association, Berkeley, CA, USA, 3-3.
- [9] Cebix. Cache Line Ping-Pong. Web site: <http://everything2.com/title/cache+line+ping-pong>
- [10] Adve, S. CS433g final exam Web site: http://www.cs.uiuc.edu/class/fa05/cs433g/assignments/Fall_2004_Final_Solution.pdf
- [11] Hyper Transport Consortium. HyperTransport 3.1 Specification. Web site: <http://www.hypertransport.org/default.cfm?page=HyperTransportSpecifications31>
- [12] Torrellas, J., Lam, H.S., Hennessy, J.L., False sharing and spatial locality in multiprocessor caches. In *Computers, IEEE Transactions*, vol.43, no.6, pp.651-663, Jun 1994. doi: 10.1109/12.286299

A RISC-Based Moving Tiny Threads Architecture

Ville Leppänen^α, Jari-Matti Mäkelä^α, Martti Forsell^β

^αDepartment of Information Technology, University of Turku, Finland,
Email: { Ville.Leppanen, jmjak }@utu.fi

^βVTT, Platform Architectures, Oulu, Finland; Email: Martti.Forsell@vtt.fi

Abstract—We describe a RISC-based architecture for moving tiny light-weight threads instead of data in the multicore context. We assume the architecture to consist of homogeneous cores that are connected with an on-chip network. A sparse 3D torus is considered as a network delivering enough bandwidth. Besides rather ordinary ALU capabilities, each core maintains a rather large set of threads, and has separate memory and instruction caches. For the data cache, we avoid coherence problems by partitioning the memory and assuming that a portion of the main memory can be accessed only via one specific data cache. Consequently, we need to move light-weight hardware-supported threads between the cores in the on-chip network.

Compared to approaches where all memory locations are accessible via each cache, we avoid coherence problems and do less loads to caches. The price is an additional network between the cores and possible inefficiencies due to moving threads between the cores. As the threads in each core are used to hide memory access and thread moving latencies, we characterize requirements for the amount of tiny threads.

1. Introduction

We describe our RISC-based multicore architectural framework designed in our MOTH project¹ for implementing a PRAM-based (Parallel Random Access Machine; [5]) approach for parallel programming. Previously this architecture has been outlined in [10]. By this architecture, we aim to provide better programmability of parallel systems, since the basis of PRAM approach is a synchronous shared memory based execution of threads. The synchronous nature of execution essentially means that there are plenty of points in the program, where the programmer can relay that the

previous memory write (and read) instructions have taken place. Consequently, the state of the program (concerning all threads) is clear and therefore designing a correctly functioning multithreaded program becomes easier. The PRAM has several variations regarding the choice of synchronization points. The most strict interpretation is that (implicit) synchronization takes place after executing a single step from all currently existing threads.

Previously, PRAM implementation has been studied in the SB-PRAM project [6] in the multicomputer context. Recently, such PRAM implementations have been provided in the multicore on chip context by Forsell [2], [1] (Eclipse architecture) and by Vishkin et al [11] (Paraleap architecture). We have also previously proposed non-RISC based solutions for the moving threads architecture [3], [4]. A simulator and a compiler [9] also exist for this RISC-based experimental architecture, but those are not discussed in this paper.

In Section 2 we present our architectural RISC-based framework for implementing the moving threads approach. We focus on the architecture of cores. The inter-core network solution is discussed in Section 3 where we also analyze efficiency requirements.

2. RISC-based architectural framework

2.1 Overview of multicore system

An overview of our architectural framework is shown in Fig. 1. The system consists of c RISC-based cores, an inter-connection network between the cores, and a main memory system. Each core maintains a set of threads, can execute instructions from those, send and receive threads via the network, and has a cache memory for accessing a part of the main memory. Each core C_i “sees” a unique fraction of the main memory via its data cache – such memory locations are called local to C_i . Thus, if a thread residing at core C_i issues a memory instruction concerning some memory

¹This research has been funded by the Academy of Finland project number 128729.

location local to core C_j , then the thread must be moved to C_j before executing the instruction. Moving a thread basically means moving the contents of its registers and program counter (as well as possibly next decoded instruction). The program, being executed by a thread to be moved, is not moved, since each core has an *instruction cache*, which contains fractions of all program codes being executed by the threads residing at that core. In this paper, we assume that the number of registers is minimal considering the RISC architecture; 2–4 registers per thread.

Each memory location is local to only one core. Thus, there are no consistency problems, since there is no real replication of the contents of memory locations. Each memory location can be cached. The data caches of cores act as root access points into the main memory. In the framework, we do not specify how the main memory is organized – e.g. it can be partitioned into blocks. We neither do not fix the organization of the memory system – there can be multiple levels of caches. The mapping of memory locations into cores is not fixed in our architectural framework. We expect such a mapping to be balanced, but leave it open whether the mapping is static or dynamically set by the executed programs.

We explain the basic function of a core next. Each core maintains a dynamically varying set of threads by storing their register values in a *register file* and maintaining other information regarding them in a *thread pool*. A core extracts instructions from the threads (by using their program counter value) in its thread pool and injects such instruction into its instruction execution *pipeline*. None of instruction in the pipeline is a non-local memory instruction. The nature of next instruction is determined at the end of execution pipeline – thus, the need to move a thread is determined as early as possible.

The goal is that each of the cores has $\Theta(X)$ threads to execute, and the threads are independent of each other – i.e. the core can take any of them and advance its execution. By taking an instruction cyclically from each thread, the core can wait for memory access taking a long time (and even tolerate the delays caused by moving the threads). The key to hide the memory (as well as network and other) delays is that the average number of threads X per core must be higher than the expected delay of executing a single instruction from any thread.

The network connecting the cores is for moving threads between the cores. Thus, each core has separate *thread buffers* for sending and receiving threads. The received threads are moved into the thread pool of the receiving core, and respectively sending means removing a thread from the pool of the sending core. The network between cores is discussed in Section 3.

The execution of all threads in the whole system is *synchronous*. The most strict interpretation of PRAM execution is that all threads execute synchronously stepwise – meaning that there is implicit synchronization after each step (i.e. atomic instruction). A less strict interpretation is that there is a separate synchronization instruction in the instruction set, and encountering such an instruction in the execution is treated as a barrier synchronization point (all threads pass over a barrier when all the threads have reached it). The instructions of a thread between two barrier synchronization points can be called as a superstep (notice that the length of superstep does not need to be static). The approach to the nature of execution synchrony is very crucial considering the semantics of programs and ease of programming. It is obvious that the more strict synchrony the easier to program but the more costly to implement. In our architectural framework, we do not specify how often the threads are synchronized, but we fix the architectural method of keeping the threads in synchrony. Our method is the *synchronization wave* method which can be seen to have been outlined already in the Fluent machine. The idea of synchronization wave is that a wave front separates two consecutive (super)steps. The wave front moves over an element (whether an interconnection node or an element related to the execution pipeline of a core) once it has arrived into the element via all input "links". Moving over a node means that the wave front is forwarded to all possible output "links" of the node.

2.2 Architecture for a single core

The bedrock of our thread processor model is a RISC-type pipeline architecture. The major change to the basic 5-stage textbook model is the the adoption of the moving threads; the address space is distributed among all cores, and a thread move occurs when the next instruction in the control flow refers to a non-local memory address or performs a special thread control instruction. This makes it necessary for an

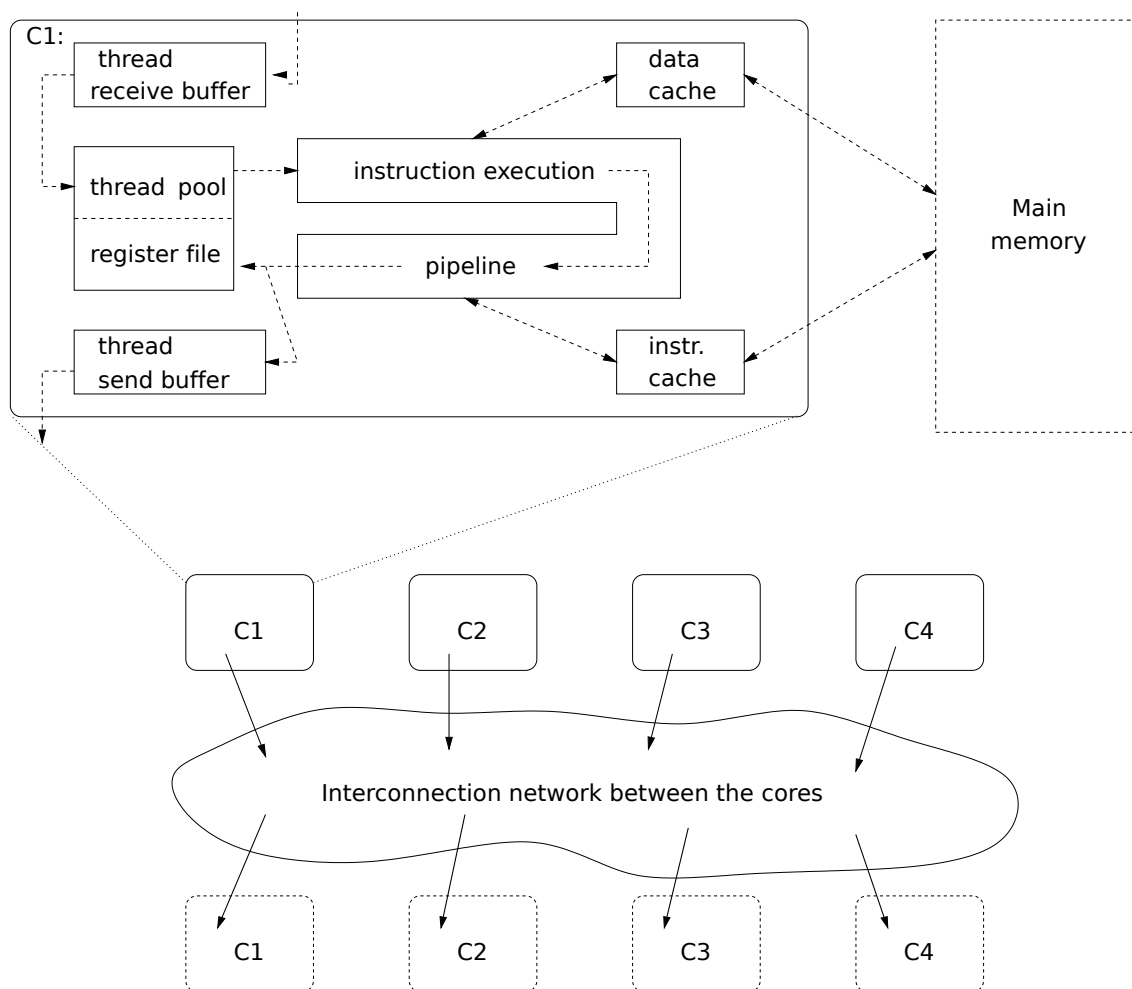


Figure 1: Overview of our multicore system.

efficient implementation to precalculate the address of a reference before execution. Another difference is the pipeline feeding technique – instead of executing instructions in coarse-grained blocks from a same thread at a time, fine-grained thread level parallelism is harnessed by alternating the executable thread between all pipeline stages. Our framework introduces a novel reorganization of the pipeline stages to fulfill both goals.

The core operational flow consist of five pipeline stages: *select*, *decode*, *execute & fetch next*, *writeback & predecode*, *address calculation & data memory access* (Figure 2). The data buffering forms the sixth stage, but it is actually an independent background task running concurrently with the main pipeline. Each of the pipeline stages have been balanced to execute in one cycle.

Ideally an instruction completion through the data path takes five cycles. In case of an instruction cache miss, a placeholder *nop* operation is executed as next instruction until the instruction becomes available. Data memory misses are handled by the sixth stage – the thread's status remains unavailable until the data has been fetched to the data buffer.

2.2.1 Instruction selection

The first state of the data path, *select*, is responsible for selecting the next available thread for execution. The threads are organized to an indexed array of rows, the *thread table*. Each row consist of three fields; the thread's status, program counter value, and a preloaded next instruction.

The status field has seven possible states (Figure 3); *free*, *ready*, *exec*, *wait*, *sync*, *move* and *recv*. The *free*

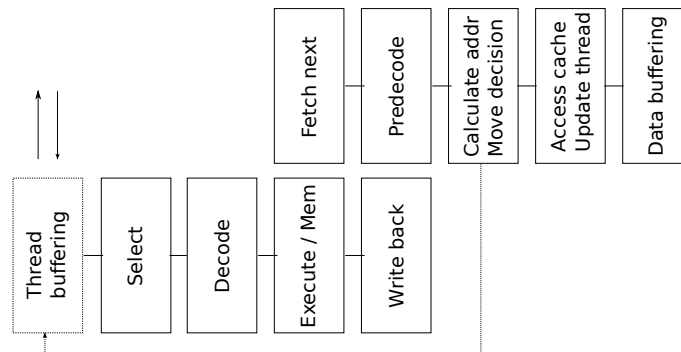


Figure 2: The pipeline stages of the thread processor core

state indicates an empty slot in thread table where a new thread can be assigned. A thread is ready for selection and execution when its state is *ready*, whereas the *exec* state means that thread's current instruction is under processing. The *wait* state expresses pending memory request from the data memory. Threads with the state *sync* wait for the next synchronization point before continuing execution. Finally, while the table control logic is receiving or moving away the thread in the slot, states *recv* and *move* are used respectively.

The select and table control unit searches on every cycle the thread table for threads with the status *ready*. If a thread is available, its status is updated to *exec*, and the prefetched instruction, program counter, and row index values are fed to the next pipeline stage (Figure 4). The program counter value is increased by four (word aligned instruction memory) before pipeline registers. If no threads are available, a *nop* operation along with an illegal row id value (discarded in later stages) is provided instead.

We assume a simplified single stage thread table model. A more realistic implementation may face problems with the speed of lookups. Those problems could be tackled by e.g. banking the table and/or building a tree like selection logic for the row id values of *ready* threads.

2.2.2 Instruction decoding

On every cycle the instruction decoder decomposes the thread's next prefetched instruction into an immediate value and possible register references, and issues respective register file fetches. A new program counter value is precalculated for possible branching and an immediate value for the ALU. The instruction data, program counter, row index, and relevant register and

immediate values are fed to the next stage.

All pipeline stages assume that the operations on the register file take exactly one cycle. Additionally, the decoding stage requires two dedicated read ports for ordinary instructions and four ports if instructions MADD and MSUB are supported. In case support for vector (SIMD) operations is desired, the total number of register file read/write ports is dictated by the widest vector operations. The predecode stage and thread move mechanism require additional dedicated ports.

2.2.3 Execution and fetching next

The third stage performs three operations every cycle: execution of the instruction, fetching of the next instruction from the instruction cache, and fetching of data from the data buffer. The data buffer is guaranteed to contain the correct data in case the instruction is a load since the execution is only triggered by a successful load. Otherwise the result is discarded in the following stages.

The execution of the instruction is performed by the ALU using the precalculated register and immediate values. The result along with the value of the second register are fed to the next pipeline stage. The data fetched from the read port of the data buffer and the instruction from the instruction cache are pushed forward to the next pipeline stage.

The data buffer is concurrently updated via two ports connected to pipeline registers. The pipeline registers store (data, row id) pairs coming from the data cache. The two port approach allows updating the buffer every cycle despite its single cycle latency. A failed instruction cache fetch results in a *nop* operation, and the program counter value is decreased by four. In other cases, the program counter and row id values are

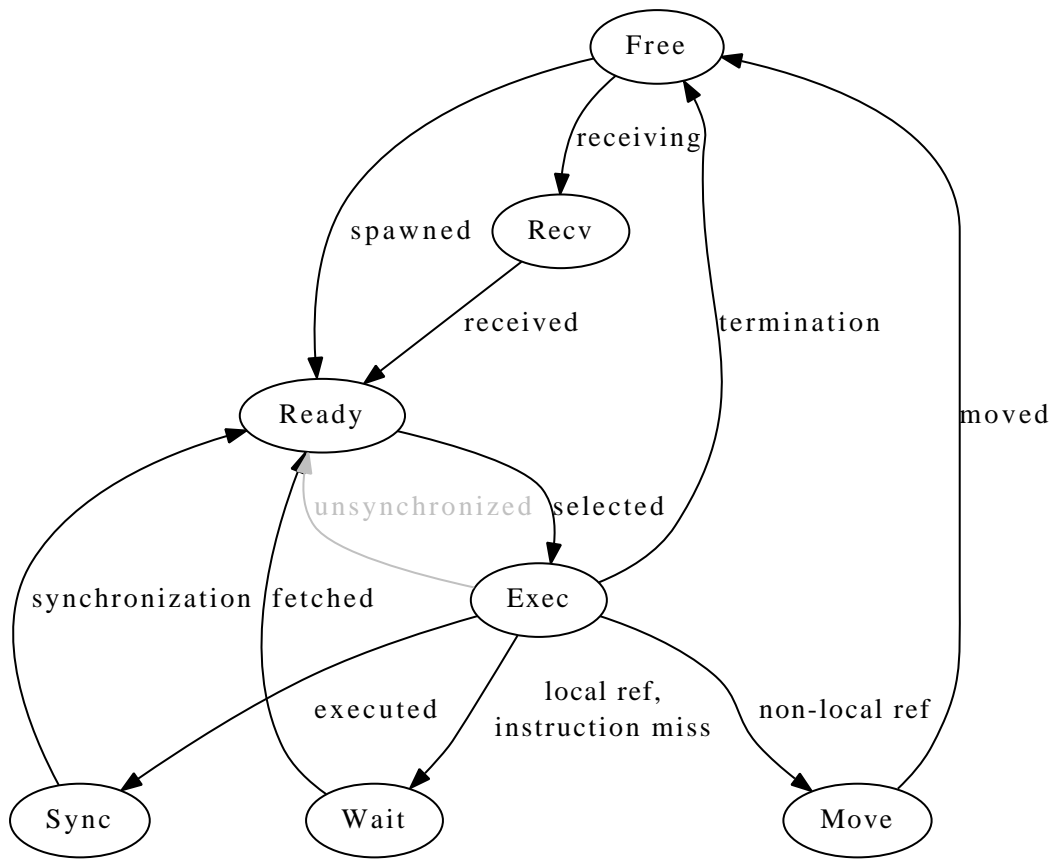


Figure 3: The flowchart of the thread states.

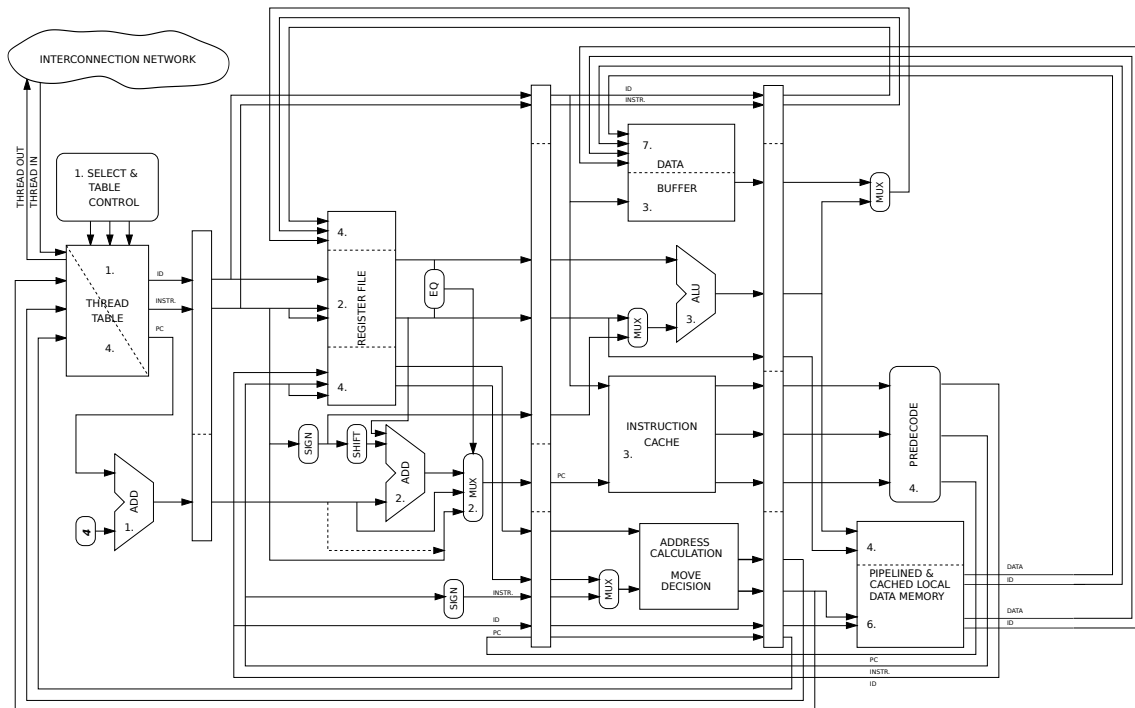


Figure 4: The datapath model of the thread processor core.

simply passed through. The original instruction data is also passed to the next pipeline stage.

2.2.4 Write back and predecoding

The fourth stage finishes the execution of the instruction by writing the register value back to the register file and storing the computed data or register value to a memory location. The original instruction data is used to determine this operation. Both write operations are assumed to have a latency of one cycle – internal processing may proceed asynchronously until the next synchronization point. Two dedicated register file write ports are required by the 32-bit *MULT* and *DIV* instructions, otherwise a single port is sufficient.

The rest of this stage deals with the predecoding of the next instruction. The instruction is decomposed as in the decode stage into the opcode and the register references. A single register file read port is reserved for fetching a register in case the next instruction is a load or store and uses a register value when computing the next address. The instruction data along with the potential fetched register value are fed to the next pipeline stage.

2.2.5 Calculating address and data memory access

The last stage of the pipeline checks whether the next instruction is accessing the memory local to the core or if there is a need to migrate the thread to another core. The mechanism used to determine the correct core for the memory reference can be as simple as a hard coded pair of hash function and the core's number. The updated thread state is written back to the thread table.

If the next instruction does not access memory or accesses a local address, the computed address is propagated to the data cache's queue and the thread's status is updated to *sync*. In other case a thread move is initiated by setting the thread's status to *move*. The quadruple of row id, thread status, next instruction, and program counter value are used to update the row in the thread table. The update and selection in the first stage use different rows and can be issued concurrently.

The select and table control unit initiates a possible thread move concurrently in the background. The row to be moved will remain untouched by the thread selection and its status is later updated to *free* when the thread's register values have been completely copied to the transmission queue. The details of thread move

have been omitted from the Figure 4.

2.2.6 Data buffering

The data memory requests performed in stage 5 are added to the data cache's queue. After processing the query, the cache stores the result in one of its two pipeline registers along with the accompanying thread row in the thread table. The data buffer unit reads these values via two input ports and updates the values in a single cycle. The data cache also signals the select and table control unit with the thread's row id associated with the data. The unit then updates the status of the row from *wait* to *ready*. The latency associated with this operation depends on where the data is fetched from the memory hierarchy.

3. Inter-core network and efficient execution requirements

Many kinds of sparse networks can be used as the inter-core network [8], e.g. a sparse butterfly, a sparse mesh, a sparse torus, etc. While the sparse networks have different properties concerning scalability, physical connection length and degree of nodes, they all share a property: a c -core sparse network can accept $\Theta(c)$ new messages per step and deliver $\Theta(c)$ messages to their targets per step. The delivery of a message involves a delay comparable with the diameter ϕ of the network, and ϕ can be non-modest. In traditional approaches, the messages correspond to read or write requests and replies, whereas in the moving threads approach, a message moves a thread consisting of a program counter, an id number, and a small set of registers. The messages in the moving threads approach are a little bit longer, but respectively there is no need for a network deliver the replies of read requests.

As we use the synchronization wave technique, we must assume the network between the send buffers and the receive buffers to consist of a number of intermediate nodes whose connections form a DAG (from the send buffers to the receive buffers). We assume the throughput of the network to be such that each node can send and receive a thread approximately every δ cycles. We denote by T_L the average latency (in cycles) of moving a thread in the network from one core to some other core. Moreover, we assume that δ is some small constant, independent of c .

The sparse 3-dimensional directed torus is such a DAG network. Its layout properties are discussed in [8]

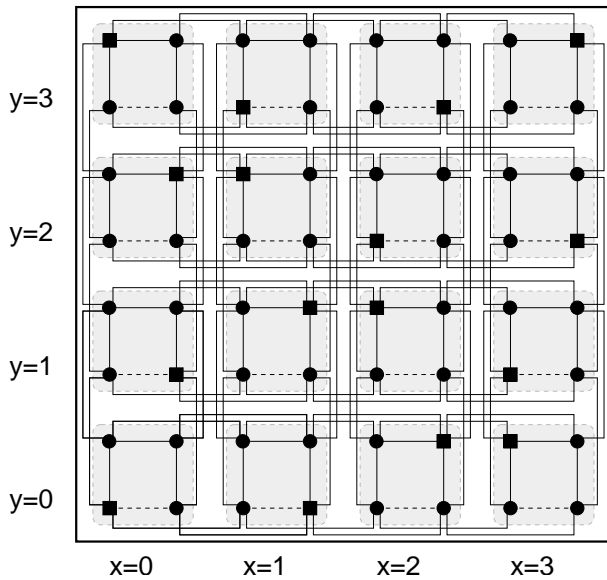


Figure 5: A grid-noc layout of a 3-dimensional sparse mesh. All connections between the nodes are shown. Boxes are cores and circles are intermediate nodes.

(a grid-noc layout of related 3D sparse mesh is shown in Figure 5). The c -core 3-dimensional sparse mesh has $c^{3/2} - c$ intermediate nodes. The connections between cores are of constant length (two core-nodes), the node in- and out-degrees are 3, and the diameter is $3\sqrt{c}$. The network can move $3c^{3/2}$ packets per step, thus the network capacity is $\delta \times 3/2$ -fold compared to the data moving requirements. As δ is often ≈ 3 for RISC-architectures, it can be easily concluded that the routing capacity will not cause any congestion to influence on T_L . Thus, we consider that $T_L \approx 2\sqrt{c}$, if the targets of thread movement operations are reasonably evenly distributed. The trick to achieve this in connection of EREW is well known: The shared address space is distributed with a hash function.

The remaining efficiency requirement is that the average number of threads per core X is higher than the latency, that is $X > 2\sqrt{c}$. This is the real price of the moving threads architecture, since each core must physically support ρX threads (for some $\rho > 1$) and the programs must the task for at least $c \times X$ parallel threads. However, this should be possible as a work-optimal polylogarithmic PRAM algorithm is known for many problems with over-linear work-complexity.

One should notice that even e.g. for $c = 10000$ cores this means that a core needs to support at least 200 threads – supporting e.g. 1000 tiny threads with only

2–4 registers seems not an unrealistic assumption.

4. Conclusions

In this paper, we characterized a RISC-based architecture for moving tiny threads to implement the classical PRAM approach. Each executed thread is modeled as a 2–4 registers, an execution state information, a program counter, and a core-related unique id number. The 3-dimensional torus network was calculated to be suitable inter-core network for the architecture. It is scalable, constant-degree, constant wirelength network, whose latency hiding requires support for rather modest amount of threads core-wise.

References

- [1] 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, 39–64, 2010.
- [2] M. Forsell. A scalable high-performance computing solution for network-on-chips. *Micro, IEEE*, 22(5):46 – 55, sep–oct 2002.
- [3] M. Forsell and V. Leppänen. Supporting Concurrent Memory Access and Multioperations in Moving Threads CMPs. In *Proceedings of PDPTA 2010*, pages 377–383, 2010.
- [4] M. Forsell and V. Leppänen. Moving Threads Processor Architecture. *Journal of Supercomputing*, page To appear, 2011.
- [5] S. Fortune and J. Willie. Parallelism in Random Access Machines. In *Proceedings, 10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.
- [6] J. Keller, C. Kessler, and J. Träff. *Practical PRAM Programming*. Wiley, 2001.
- [7] V. Leppänen. “Balanced PRAM Simulations via Moving Threads and Hashing.” *Journal of Universal Computer Science*, 4:8, 675–689, 1998.
- [8] V. Leppänen, M. Penttonen, and M. Forsell. Layouts for Sparse Networks Supporting Throughput Computing. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2010)*, pages 443–449, 2010.
- [9] J.M. Mäkelä and V. Leppänen. Towards programming on the moving threads architecture. In *CompSysTech '10: Proceedings of the International Conference on Computer Systems and Technologies*, pages 137–142. ACM Press, 2010.
- [10] J. Paakkulainen, J.M. Mäkelä, V. Leppänen, and M. Forsell. Outline of risc-based core for multiprocessor on chip architecture supporting moving threads. In *CompSysTech '09: Proceedings of the International Conference on Computer Systems and Technologies*, pages 1–6. ACM Press, 2009.
- [11] X. Wen, U. Vishkin. “FPGA-based prototype of a PRAM-On-Chip processor.” *Computer Frontiers 2008*, May 5-7, 20

Parallel RISC Architecture.

A Functional Approach Based on Backus's FP language

Mihaela Malița¹, and Gheorghe M. Ștefan²

¹Saint Anselm College, Manchester, NH, (mmalita@anselm.edu)

²PUB, Bucharest, Romania, (gstefan@arh.pub.ro)

Abstract – *The main consequence of building ad hoc structured hardware for parallel computation is the huge difficulty we have to program it. The paper discusses a new framework introducing the concept of **parallel RISC engine**, as a simple and efficient solution for executing FP like languages proposed by John Backus [2] as an alternative to the von Neumann style of performing computation. A first version for the hardware solution is already implemented in silicon [12].*

Key words: parallel architecture, parallel programming, functional programming, integral parallel architecture, Backus's FP System.

1 Introduction

No one describes better the deadlock of parallel computing than David Patterson which last June wrote in [6] the following about the stage of multi-core industry:

"... the semiconductor industry threw the equivalent of a Hail Mary pass when it switched from making microprocessors run faster to putting more of them on a chip-doing so without any clear notion of how such devices would in general be programmed. ... The trick will be to invent ways for programmers to write applications that exploit the increasing number of processors found on each chip without stretching the time needed to develop software or lowering its quality. Say your Hail Mary now, because this is not going to be easy."

Indeed, parallel computing started wrong, with *ad hoc* constructs considering that more than one machine, more or less sophisticatedly interconnected, will have the brute force to solve the continuously increasing hunger for computing power. The approach was, and is, wrong for two obvious reasons:

- **programmability** is very low, because it was proved that is impossible to ignore the sophisticated physical details in order to write efficiently complex programs
- **portability** is also very low, because code already written for sequential algorithms is almost impossible to be efficiently translated automatically for various complex parallel engines

and one hidden, but essential reason:

- the lack of **parallel architecture** which is supposed: (1) to hide from the programmer the physical details of the actual engine, and (2) to facilitate the automatic translation of the huge sequential software legacy in as much as possible efficient parallel code.

Thus, the problem is to find the shortest path from an appropriate computational model to the simplest possible parallel architecture and to find a validation procedure. Our proposal is:

1. to start with Stephan Kleene's computation model of *partial recursive functions*, because it is a *n*-based model, i.e., it assumes in the initial statements the use of *n* variables and/or functions, with *n* of any size
2. to associate, as the simplest architectural interface, the *Functional Program System* (FP System) proposed by John Backus [2], because of its inherent parallel approach
3. to use "*The Berkeley's View of Parallel Landscape*" [1] as the validation environment for the simplest generic hardware implementation: a **ConnexArray**TM based engine [9].

We presented the first step in [5], the second step is initiated in this paper, while the last one will be only sketched in this paper and is left to be completed for future works. The second section describes the structure of the parallel RISC engine (pRISC), the structure

which emerged from Kleene's computation model. The third section proves that the FP system of Backus describes efficiently the architecture of the pRISC engine. Preliminary evaluations of the first embodiment of a pRISC architecture – a *ConnexArray*TM based system – are sketched in the last section.

2 The parallel RISC engine

The path from an appropriate computational model to an integral parallel architecture (IPA) is covered in [5] and the associated last silicon implementation – a 65 nm version of a SoC built by *BrightScale*, containing a *ConnexArray*TM with 1024 execution units – is described in [12]. *ConnexArray*TM is the hardware considered in our approach as the generic support for a pRISC engine. It is represented in Figure 1, where:

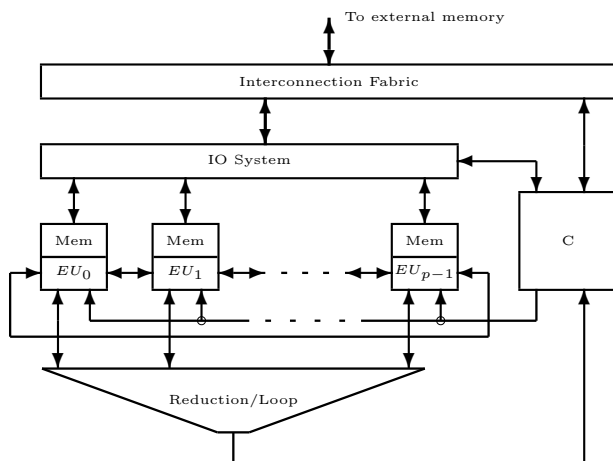


Figure 1: *ConnexArray*TM. The cellular array of p execution units – EU_0, \dots, EU_{p-1} – each with its own local memory (Mem) is controlled by a sequential engine (C).

C is a general purpose sequential processor used as the controller of the whole system. It issues the sequence of instructions executed in predicated mode in each one of the p execution units (EU).

EU_i is a small & simple EU which executes, according to its internal state, the instruction issued by C.

Mem is the local memory in each processing cell. It is used to store data (the entire cell works as an execution unit) or data & programs (the entire cell works as a processing element, PE).

Reduction/Loop is a tree structured circuit which implements: (1) vector to scalar reduction func-

tions, sending back to C the result, (2) closes a combinational loop over the array of EUs.

IO System transfers data vectors transparently to the processing performed in each one of the p cells

Interconnection Fabric controls the data & program transfers between C & array and the external memory.

The user view of *ConnexArray*TM is a two-dimension array of scalars (see Figure 2): the constant vector **index**, m p -component vectors stored in the cellular distributed memory (each *Mem* module stores m scalars) and t p -component vectors distributed in the EU's register files.

index	1	2	p
v_1	s_{11}	s_{12}	s_{1p}
v_2	s_{21}	s_{22}	s_{2p}
\vdots	\vdots	\vdots	\vdots	\vdots
v_m	s_{m1}	s_{m2}	s_{mp}
r_{u+1}	r_{11}	r_{12}	r_{1p}
\vdots	\vdots	\vdots	\vdots	\vdots
r_{u+t}	r_{t1}	r_{t2}	r_{tp}

Figure 2: The user view of *ConnexArray*TM. The local memories *Mem* store m p -component vectors, while the local register files store t p -component vectors. The horizontal (spatial) dimension provides time consuming communication, while the vertical (temporal) dimension allow any efficient virtual interconnection network.

The user of *ConnexArray*TM sees a $p \times m$ array of scalars. The horizontal (spatial) dimension p is supported by a very simple linear interconnection network, i.e., the distance between two elements on this dimension is in $O(p)$. The simple & small interconnection hardware implies low speed on this dimension. The vertical (temporal) dimension m is supported by the most flexible "interconnection network": the random access mechanism of the *Mem* modules in each EU. On this dimension the distance between two elements is small & constant. The vertical flexibility can and must be used in order to deal with "big distance" connections in the two-dimension array of variables.

The controller C has a standard organization centered on a register file of u 32-bit scalars. The instruc-

tion set executed by the whole system – C & EUs – is defined on the concatenation of two register files:

- the scalar register file of C
- the vector register file distributed in the p EUs

thus, the instruction set is defined on $t + u$ registers, the first u registers are the scalar registers in C, while the next t registers are vector registers in the p EUs. For example, let be $t = u = 16$. Then, the instruction

```
add r24 r3 r27;
```

adds to each component stored in the vector register 27 the value from the **scalar** register 3 and sends the result in the vector register 24, while the instruction

```
add r24 r18 r27;
```

adds in **r24** the **vector r18** with vector **r27**.

The specific instruction for vector processing in *ConnexArray*TM is the predicated execution expressed as in the following example:

```
where (r25 == 0) add r20 r20 17 ;
elsewere xor r20 r20 r20 ;
```

where: in all the cells **where** the component of the vector stored in the register 25 is zero, the component of the vector stored in the register 20 is incremented with 17, while **elsewere** (where the component of the vector stored in the register 25 is different from zero) the component of the vector stored in the register 20 is cleared.

Another specific function is:

```
where (r20 = <1,5,6>) first r22;
```

which provides in **r22** a vector with 1 only on the first position pointed by the sub-vector <1,5,6> in **r20**, and 0 in rest. For example: if

```
r20 = <7,4,1,5,7,1,5,6,8,1,5,6,4,2,4,5>
```

then the result is

```
r22 = <0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0>.
```

Those kinds of operations are supported by the combinatorial loop performed by *Reduction/Loop* circuit.

The full power of the pRISC engine must be *proved* using a theoretical programming model and *evaluated* using the broadest possible functional spectrum.

3 FP system on pRISC

The seminal paper of John Backus *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs* [2] offers the best theoretical environment for proving that the pRISC

engine is a good candidate for a generic parallel computer. The Functional Programming Systems (FP Systems) proposed by Backus was introduced as an alternative to the *von Neumann style of programming* – the main paradigm of sequential computation. **FP Systems can be seen as the definition of parallel systems from the programming point of view.** Therefore, we consider that the pRISC engine, defined in the previous section, must be able to deal efficiently with the *objects, functions* and *functional forms* introduced by Backus in his FP Systems.

Objects. The objects defined in FP Systems, *atoms* and *sequences*, can be found in the pRISC architecture in the form of scalars (x, y, \dots) managed by the controller C and the vectors ($\langle x_1, \dots, x_p \rangle$) processed in the linear array of EUs.

Functions. The following set of primitive functions, almost identical with the set proposed in [2], are examined from the point of view of how they are implemented in our pRISC engine.

Selector:

$$i : \langle x_1, \dots, x_p \rangle \rightarrow x_i$$

The value i is searched in the constant vector **index** (Figure 2), the resulting Boolean vector is used to select the component x_i and to send it to the controller C through the Reduction circuit.

Tail:

$$tl : \langle x_1, \dots, x_p \rangle \rightarrow \langle x_2, \dots, x_p \rangle$$

The first element is searched using the **index** vector and is deleted using the serial connections between EUs.

Reverse:

$$reverse : \langle x_1, \dots, x_p \rangle \rightarrow \langle x_p, \dots, x_1 \rangle$$

This is the most difficult function for the pRISC engine, because of the very simple interconnection network. It is performed with the help of the IO System, which is featured with hardware support for performing any permutation as an *out of core* function.

Distribute:

$$distrl : \langle y, \langle x_1, \dots \rangle \rangle \rightarrow \langle \langle y, x_1 \rangle, \dots, \langle y, x_p \rangle \rangle$$

$$distr : \langle y, \langle x_1, \dots \rangle \rangle \rightarrow \langle \langle x_1, y \rangle, \dots, \langle x_p, y \rangle \rangle$$

are solved depending on the computational context in two ways: (1) by loading an uniform vector containing on each position the value y , or (2) by issuing a $\{scalar, vector\} \rightarrow vector$ operation with the scalar y broadcasted by C toward each EU.

Length:

$$length : \langle x_1, \dots, x_p \rangle \rightarrow m$$

Is performed in two steps: (1) generates a Boolean vector with 1 on each position corresponding to the p components of the vector, and (2) using the **add** reduction function, the 1s are added and sent to the C controller.

Transpose:

$$\begin{aligned} trans : &<< x_{11}, \dots, x_{1m} \rangle, \langle x_{21}, \dots, x_{2m} \rangle, \dots, \\ &\langle x_{n1}, \dots, x_{nm} \rangle \rangle \rightarrow \\ &<< x_{11}, \dots, x_{n1} \rangle, \langle x_{12}, \dots, x_{n2} \rangle, \dots, \\ &\langle x_{1m}, \dots, x_{nm} \rangle \rangle \end{aligned}$$

is solved on the temporal dimension with no computation because expanding each of the m variables of the initial vectors on the spatial dimension (horizontally), we obtain the n variables of the final vectors vertically, on the temporal dimension:

$$\begin{aligned} &\langle x_{11}, \dots, x_{1m} \rangle \\ &\langle x_{21}, \dots, x_{2m} \rangle \\ &\dots \\ &\langle x_{n1}, \dots, x_{nm} \rangle \end{aligned}$$

where each vector is a m -variable "column".

Append:

$$\begin{aligned} apendl : &\langle y \langle x_1, \dots, x_p \rangle \rangle \rightarrow \langle y, x_1, \dots, x_p \rangle \\ apendr : &\langle y \langle x_1, \dots, x_p \rangle \rangle \rightarrow \langle x_1, \dots, x_p, y \rangle \end{aligned}$$

Is solved inserting in the first position the scalar y issued by the controller C.

Rotate:

$$\begin{aligned} rotl : &\langle x_1, \dots, x_p \rangle \rightarrow \langle x_2, \dots, x_p, x_1 \rangle \\ rotr : &\langle x_1, \dots, x_p \rangle \rightarrow \langle x_p, x_1, \dots, x_{p-1} \rangle \end{aligned}$$

The two-direction linear connection between cells allows the rotate function in both directions. For vectors with less components than the number of EUs the reduction and insert function are used to perform the operations.

Search:

$$\begin{aligned} src : &\langle y, \langle x_1, \dots, x_p \rangle \rangle \rightarrow \langle b_1, \dots, b_p \rangle \\ \text{with } b_i &= (y = x_i) ? 1 : 0. \end{aligned}$$

The scalar y is issued by the controller and is searched in each EU generating a Boolean vector with 1 on each match position.

Conditioned search:

$$\begin{aligned} csrc : &\langle y, \langle x_1, \dots, x_p \rangle, \langle b_1, \dots, b_p \rangle \rangle \rightarrow \\ &\langle c_1, \dots, c_p \rangle \\ \text{with } c_i &= ((y = x_i) \& b_{i-1}) ? 1 : 0. \end{aligned}$$

The search is performed only in the cells preceded by a cell when the previous search (src or csrc) provided a match.

Example. The sequence of operations:

src d, csrc o, csrc g, csrc _

identify all the occurrences of the sequence dog_ in the sequence $\langle x_1, \dots, x_p \rangle$. It allows to define the **stream search** operation.

Stream search:

$$ssrc : \langle \langle y_1, \dots, y_s \rangle, \langle x_1, \dots, x_p \rangle \rangle$$

returns a Boolean vector with 1s pointing all the occurrences of $\langle y_1, \dots, y_s \rangle$ in $\langle x_1, \dots, x_p \rangle$.

Insert data:

$$\begin{aligned} ins : &\langle x, k, \langle x_1, \dots, x_p \rangle \rangle \rightarrow \\ &\langle x_1, \dots, x_{k-1}, x, x_k, \dots, x_p \rangle \end{aligned}$$

The scalar k is searched in the **index** vector to identify the insert position for x .

Delete:

$$\begin{aligned} del : &\langle k, \langle x_1, \dots, x_p \rangle \rangle \rightarrow \\ &\langle x_1, \dots, x_{k-1}, x_{k+1}, \dots, x_p \rangle \end{aligned}$$

The scalar k is searched in the **index** vector to identify the delete position.

Functional forms. A functional form depends of functions or objects. The most common functional form is the **composition**, which denotes the application of a sequence of functions f_1, f_2, \dots, f_q to x :

$$(f_1 \circ f_2 \circ \dots \circ f_q) : x \equiv ((f_1 : f_2 : \dots : (f_q : x) \dots))$$
Construction:

$$[f_1, \dots, f_n] : x \rightarrow \langle f_1 : x, \dots, f_n : x \rangle$$

Is parallel speculation. It can be performed on the variable x issued by C and processed in each EU according to the content (data and/or program) of the local memory *Mem*.

Insert:

$$/f : \langle x_1, \dots, x_p \rangle \rightarrow f : \langle x_1, /f : \langle x_2, \dots, x_p \rangle \rangle$$

Is executed as a reduction function in $O(\log p)$ time.

Apply to all:

$$\alpha f : \langle x_1, \dots, x_p \rangle \rightarrow \langle f : x_1, \dots, f : x_p \rangle$$

Is typical data parallelism. The function f is stored in the program memory used by C.

Condition:

$$\begin{aligned} (p \rightarrow f; g) : &\langle x_1, \dots, x_p \rangle \rightarrow \\ &\langle (p \rightarrow f; g) : x_1, \dots, (p \rightarrow f; g) : x_p \rangle \end{aligned}$$

where:

$$(p \rightarrow f; g) : x \rightarrow \text{if } ((p : x) = 1) f : x; \text{ else } g : x;$$

Is the data parallel predicated execution.

Example. Let be the following definition:

Def $ABS \equiv (/+) \circ (\alpha(lt \rightarrow (- \circ reverse); -)) \circ trans$

Applying it:

$$ABS : \langle \langle x_1, \dots, x_p \rangle, \langle y_1, \dots, y_p \rangle \rangle$$

provides the sum of absolute difference of the two vectors: $\langle x_1, \dots, x_p \rangle$ and $\langle y_1, \dots, y_p \rangle$.

The functions used by Backus to define FP Systems are efficiently executed by the pRISC engine. The associated architecture is expressed as a FP System. This approach represents a theoretical backup for the specification of a functional language for pRISC like engines.

Thus, parallel computation can start based on a solid theoretical foundation, avoiding risky *ad hoc* constructs. The pRISC engine and the associated FP System based architecture, complemented with multi-threaded hardware support (see [13]), is a promising start in saving us from saying "Hail Mary" when deciding what to do to improve our computing machines.

4 Programming pRISC in FP

The four forms of parallelism allowed by the pRISC architecture – data, time, speculative and reduction parallelism (see [13]) – cover theoretically all aspects of the intense computation paradigm [12]. But, the efficiency of pRISC in performing all the aspects of intense computation remains to be proved. In this section we sketch only the complex process of evaluating the proposed pRISC architecture. The best plan for this process is to consider all dwarfs (motifs) outlined in “A View from Berkeley” [1], where is provided a comprehensive presentation of the problems to be solved by the emerging actor on the computing market: the ubiquitous parallel paradigm. Many decades just an academic topic, “parallelism” becomes an important actor on the market after 2001 when the clock rate race stopped. This research report presents 13 computational motifs which cover the main aspects of parallel computing. In this section we will make a preliminary evaluation of them in the context of organization and architecture just introduced in the previous two sections. The cellular network of PEs or EUs has the simplest possible interconnection network. This is both an advantage and a limitation. On one hand, the area of the system is minimized, and it is easy to hide the associated organization from the user, with no loss in programmability or in the efficiency of compilation. On the other hand, some limitations are expected in certain application domains. Follows short comments about how the proposed pRISC architecture works for all of the 13 motifs.

Dense linear algebra. The computation in this domain operates mainly on $N \times M$ matrices. The main operations performed are: matrix addition, scalar multiplication, transposition of a matrix, dot product of vectors, matrix multiplication, determinants of a matrix, Gaussian elimination, solving systems of linear equations and the inverse of a $N \times M$ matrix. Depending on the size of the product $N \times M$ the internal representation of the matrix is decided. If the product is small enough (usually, no bigger than 128), each matrix can be expanded as a vertical vector and associated to one EU, resulting in p matrices represented by $N \times M$ p -component vectors. But, if the product $N \times M$ is big, then q EUs are associated with each matrix, resulting in parallel processing of p/q matrices represented in $N \times M/q$ p -component vectors. For all the operations above listed the computation is usually accelerated almost p times, but not under $(p/(\log_2 q))$ times. The most used operation is the inner product of two vectors. It is expressed in FP System as follows:

$$\text{Def } IP \equiv (/+) \circ (\alpha \times) \circ trans$$

Sparse linear algebra. There are two types of sparse matrices: (1) randomly distributed sparse arrays (represented by few types of lists), (2) band arrays, represented by a stream of short vectors.

For small random sparse arrays, converting them internally into dense array is a good solution. For big random sparse arrays the associated list is operated using the efficient search operations provided by pRISC architecture. Thus, the multiplication of a sparse $N \times M$ matrix with a M -component sparse vector is done in $O(u + v)$, where u is the number of non-zero components in the initial vector and v is the number of non-zero components in the resulting vector.

The band arrays are first transposed using the function `trans` in a number of vectors equal with the width w of the band. Then the main operations are very easy performed using appropriate `rotl` and `rotr` operations. Thus, the multiplication of two band matrices is done on pRISC in $O(w)$.

Spectral methods. The typical examples are: FFT or wavelet computation. Because of the “butterfly” data movement, how the FFT computation is implemented depends on the length of the sample. The spatial and the temporal dimensions of the proposed architecture help the programmer to easily adapt the data representation to result in an almost linear acceleration. In order to reduce, almost eliminate, the slowdown caused by the rotate operations, the stream of samples is loaded using, as much as possible, the temporal dimension of the architecture. In [3] the FFT computation is evaluated on the pRISC architecture (for example: if FFT is considered for 1024 floating point samples the computation is done in 1 clock cycle per sample).

N-Body method. This method fits perfectly on the proposed architecture, because for $j = 0$ to $j = n - 1$ the following equation must be computed:

$$U(x_j) = \sum_i F(x_j, X_i)$$

Each function $F(x_j, X_i)$ is computed on a single EU, and then the sum is a *reduction* operation linearly accelerated by the array. Depending on the value of n , the data is distributed in the processing array using the spatial dimension only, or for large n , both the spatial and the temporal dimension are used. For this motif results an almost linear acceleration.

Structured grids. The grid is distributed on the two dimensions of our array: the spatial dimension and the temporal dimension. Each processor is assigned a column of nodes (on the temporal dimension). It performs

each update step locally and independently of other lines of nodes. Each node has to communicate only with a small number of neighboring nodes on the grid, exchanging data at the end of each step. The system works as a cellular automaton. The computation is accelerated linearly on the proposed architecture.

Unstructured grids. Unstructured grid problems are updates on an irregular grid, where each grid element is updated from its neighbor grid elements. Parallel computation is disturbed by the non-uniformity of the data distribution. In order to solve the non-uniformity problem a preprocessing step is required to generate an easy manageable representation of the grid. We expect moderate performances on pRISC.

Map reduce. The typical example of a map reduce computation is the Monte Carlo method. This method consists in many completely independent computations working on randomly generated data. This type of computation is highly parallel. Sometimes it requires the add reduction function, for which the proposed architecture has special accelerating hardware. The computation is linearly accelerated.

Combinational logic. There are a lot of very different problems falling in this class. We list here only the most important and the most frequently used:

- block processing, exemplified by AES and DES encryption. For example, AES works in 4×4 arrays of bytes, each array is loaded in one EU, and the processing is completely SIMD-like with linear acceleration on the pRISC architecture.
- recursive & non-recursive convolution encoding are computed efficiently using (1) right pipeline propagation in the array, (2) predicated data parallel processing, (3) reduction add function.
- image rotation for black & white or color bit mapped images is performed (1) by loading the $m \times m$ array of pixels into the processing array on both dimensions (spatial and temporal), (2) executing a local transformation, and third restoring the transformed image in the appropriate place.
- route lookup, used in networking; it supposes three data-base like operations: longest match, insert, delete; for all we have functions in the pRISC architecture (similar with: `src`, `csrc`, `ins`, `del`).

Graph traversal. The array of 1024 machines can be used as a “speculative device”. Each EU starts with a

full graph stored in its data memory, and the computation provides the result when one EU, if any, finds the solution. Limitations are generated by the dimension of the data memory of each EU or by the IO System capabilities. More investigation is needed to evaluate the actual power of pRISC in solving this problem.

Some problems related with graphs are easily solved if matrix computation is involved (example: computing the distance between all the elements of a graph).

Dynamic programming. Viterbi decoding is a typical example presented in [1]. The parallel strategy is to distribute the states among the cells. Each state has its own distinct cell. The inter cell communication is done in a small neighborhood. Each cell receives the stream of data which is thus submitted to a speculative computation. The work done on each processor is similar. The last stage is performed using the functions of the Reduction circuit. The degree of parallelism is limited to the number of state considered by the algorithm.

Back-track and branch & bound. The basic backtracking SAT algorithm, for example, runs on a p -cell engine by choosing $\log_2 p$ literals, instead of one on a sequential machine, assigning for them all the values from $00 \dots 0$ to $11 \dots 1$, simplifying the formula and then recursively checking if the simplified formula is satisfiable.

For parallel branch & bound we use the case of the Quadratic Assignment Problem. The problem deals with two $N \times N$ matrices: $A = (a_{ij})$, $B = (b_{kl})$. The global cost function:

$$C(p) = \sum_i^n \sum_j^n a_{ij} \times b_{p(i)p(j)}$$

must be minimized finding the permutation p of the set $N = \{1, 2, \dots, n\}$. Dense linear algebra methods already discussed are involved here.

Graphical models. Besides the Viterbi algorithm (already discussed) used for decode, this motif is well represented by parallel hidden Markov models. The architectural features reported in research papers refers to fine-grained SIMD processor arrays connected to each node of a coarse-grained PC-cluster. Thus, a pRISC engine can be used efficiently as an accelerator for general purpose sequential engines.

Finite state machine. The authors of [1] claim that for this motif “nothing helps”. But, we consider that the array of cells with their local memory loaded with non-deterministic FSM descriptions work very efficient as a speculative engine for applications such as deep packet inspection, for example.

At the end of this short introductory analysis, *which must be detailed by future investigations*, we claim that for almost all the computational motifs the pRISC architecture performs very well. Maybe for some of the motifs additional multi-threaded computation may be helpful (see [13]).

5 Concluding remarks

The pRISC engine is area and energy efficient. The architecture described and partially evaluated in this paper is based on actual silicon implementations used to measure real performances for intense computations. The hardware results are spectacular (6.5 GOPS/mm² and 40 GOPS/Watt [12]) and, complemented by the architectural support provided in this paper by a Backus's FP System approach, positions pRISC engine & architecture as a promising solution.

pRISC engine & its architecture are both small & simple. The simplicity of the engine allows its hiding behind an efficient architecture. No complex interconnection network between cells and small & simple EUs are the main premisses for a transparent architecture. "*Small is Beautiful*" claims [1]. (See also [8].)

Portability & programmability is high for a simple & generic architecture. High diversity and complexity dominate the main parallel architecture targeted today by the application engineers. What they need is *only one simple architecture* for porting existing applications or for developing new ones. The pRISC environment is a promising candidate.

Acknowledgments The authors got a lot of support from the main technical contributors to the development of the *ConnexArrayTM* technology, the *BA1024* chip, the associated language, and its first application: E. Altieri, F. Ho, B. Mițu, M. Stoian, D. Thiebaut, T. Thomson, D. Tomescu.

References

- [1] K. Asanovic, et. al.: *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report No. UCB/EECS-2006-183.
- [2] J. Backus: "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM*, August 1978, 613-641.
- [3] I. Lorentz, M. Malita, R. Andonie: "Fitting FFT onto an Energy Efficient Massively Parallel Architecture", *The Second International Forum on Next Generation Multicore / Manycore Technologies*, June, 2010.
- [4] M. Malița, G. Ștefan, D. Thiebaut: "Not Multi, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation" in *ACM SIGARCH Comp. Arch. News*, Vol. 35, 5, Dec. 2007.
- [5] M. Malița, G. Ștefan: "On the Many-Processor Paradigm", *Proceedings of the 2008 World Congress in Computer Science, Computer Engineering and Applied Computing*. vol. PDPTA'08, 2008.
- [6] D. Patterson: "The Trouble with Multicore", *IEEE Spectrum*, July 2010.
- [7] G. Ștefan, D. Thiebaut: "Memory Engine for the Inspection and Manipulation of Data", *United States Patent 6,760,821*, July 6, 2004; Filed: Aug. 10, 2001.
- [8] G. Ștefan, M. Malița: "Granularity and Complexity in Parallel Systems", *Proceedings of the 15 IASTED International Conf*, 2004, Marina Del Rey, CA, pp.442-447.
- [9] G. Ștefan: "The CA1024: A Massively Parallel Processor for Cost-Effective HDTV", in *Spring Processor Forum: Power-Efficient Design*, May 15-17, San Jose, CA 2006.
- [10] G. Ștefan, et al.: "The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing", in *Hot Chips: A Symposium on High Performance Chips*, Stanford Univ., August, 2006.
- [11] G. Ștefan: "The CA1024: SoC with Integral Parallel Architecture for HDTV Processing", in *4th International System-on-Chip (SoC) Conference and Exhibit*, November, Newport Beach, CA, 2006.
- [12] G. Ștefan: "One-Chip TeraArchitecture", *Proceedings of the 8th Applications and Principles of Information Science Conference*, Okinawa, Japan on 11-12 January 2009.
- [13] G. Ștefan: "Integral Parallel Architecture in System-on-Chip Designs". *The 6th International Workshop on Unique Chips and Systems*, Atlanta, GA, USA, December 4, 2010, 23-26.

Mobile Process Resumption In Java Without Bytecode Rewriting

Matthew Sowers (sowers@unlv.nevada.edu)

Jan Bækgaard Pedersen (matt.pedersen@unlv.edu)

School of Computer Science, University of Nevada Las Vegas, Las Vegas, Nevada, United States

Abstract—In this paper we describe an implementation of mobile processes with polymorphic interfaces in the ProcessJ language. ProcessJ is a process oriented language based on CSP and the π -calculus. In the paper we demonstrate the translation of ProcessJ to Java/JCSP and illustrate how to implement mobile processes with polymorphic interfaces without rewriting bytecode; this requires some clever code generation in Java since it does not support polymorphic interfaces.

Keywords: ProcessJ, Process-Oriented Programming, Mobile Processes

1. Introduction

In this paper we present a technique for implementing transparent mobile processes with polymorphic resumption interfaces for the ProcessJ language in Java/JCSP.

As part of the ProcessJ compiler, we have developed a code generation technique that allows a process to suspend, and subsequently resume, in the middle of a code block. When translating the ProcessJ code to Java/JCSP, a direct translation is not possible because Java does not support process suspension/resumption natively; nor does it support polymorphic interfaces.

Unlike previous attempts at resumable processes in Java [1] which describes a technique for implementing mobile processes with just one interface, the techniques described in this paper does not require any rewriting of the compiled Java bytecode. An integral part of the technique in [1] required the compiled bytecode to be changed to add explicit jumps to the resumption point. The technique described here does not need any such rewrite. Although no goto instruction is available in Java, we achieve resumption by using nested switch statements and collected state information.

Before describing the translation approach, we provide a brief overview of ProcessJ, mobility and resumability

1.1 ProcessJ

ProcessJ is a general purpose process-oriented programming language developed at the University of Nevada Las Vegas. The process-oriented primitives in ProcessJ are based on Communicating Sequential Processes, CSP [2], and the π -Calculus [3].

```

1:  mobile proc void foo(int x, int y) {
2:      int a;
3:      B1
4:      while (B2) {
5:          int q;
6:          B3
7:          suspend resume with (int z);
8:          int w;
9:          B4
10:     }
11:     B5
12: }
```

Figure 1: Sample ProcessJ Code.

The syntax of ProcessJ is similar to that of Java. There are no classes or objects, but the expression syntax and control flow would be familiar to a Java programmer. Though the syntax is similar to Java, the semantics are similar to *occam- π* [4]. As a process-oriented language, ProcessJ is composed of *processes* that each execute in their own context similar to processes in *occam- π* [4].

Figure 1 illustrates a fairly simple ProcessJ mobile process consisting of 2 interfaces. The *original* process interface, line 1, takes two integer parameters x and y . The second interface, a *resumption* interface, takes one integer parameter z at line 7. Furthermore, a number of local variable declarations (lines 2, 5, and 8) along with 5 code blocks (lines 3, 4, 6, 9, and 11) are included. The first time *foo* is called (we refer to that as *started*), it must be passed two integer values. When the *suspend* statement (line 7) is encountered, the process temporarily suspends, and control is returned to the caller. The caller can either re-invoke the process or transmit it to another process across a channel. When *foo* is invoked for the second time, it must be with the interface defined by the *suspend resume* statement (line 7), namely with just one integer value.

It should be fairly simple to imagine a translation of this code to Java for all the lines except the *suspend resume* statement in line 7. If Java had a goto statement, the implementation could be done using gotos and a some internal state (that is the end-result of the bytecode rewriting in [1]), but transforming the code into nested switch statements to achieve this as well as handling the different interfaces is more challenging.

1.2 Mobility

A classification of Mobile Code Languages is provided in [5]. The term *Strong Mobility* applies to an agent or process that is able to suspend their execution and be sent to a separate *computational environment* (CE) where it is resumed. The CE in the case of ProcessJ is either the same Java Virtual Machine, JVM, or it could mean a separate JVM. The resumed process is in the same execution state as when it was suspended.

In ProcessJ, we also offer *transparent strong mobility* [6]. The transparency comes from the ProcessJ programmer not needing to explicitly provide code to reestablish the state of the process.

1.3 Resumability

Pedersen and Kauke provide a definition of resumability in [1]. To summarize, a process is resumable if it contains a *suspend* statement. The *suspend* statement returns control the the caller, at some later point on the same or different JVM the process is resumed at the statement following the *suspend*. Since the publication of [1], the addition of polymorphic resumption interfaces, discussed later in section 3, has added an additional statement *suspend with resume* that allows the process to be resumed with different parameters.

We need not bother with the specifics of bytecode definitions of resumability because our approach does not utilize bytecode rewriting.

It should also be noted, that the techniques described in this paper are equally well suited to mobile processes that do not have polymorphic interfaces, that is, implementing single interface mobile processes in Java can be achieved by using this approach as well.

2. Serializable Processes

A mobile process in ProcessJ is implemented as a Java class. Each mobile process class implements the Java Serializable interface. Implementing the Serializable interface allows us to transfer a process to another computational environment for resumption.

State is saved and restored by implementing processes as Serializable Java classes with all local variables rewritten as fields. Saving variables as fields is an easy and convenient way of preserving state between resumptions. The rewrite is accomplished by prefixing local variable names with a block id that makes them unique at the field level. The compiler is allowed to rewrite locals as fields because ProcessJ does not have any fields.

Storing variables as fields instead of locals is the crux that makes state restoration so simple. Rather than storing local state each time there is a suspend, state is stored every time a variable is mutated. There is also no need to restore variable values during resumption because they are already available.

```

1:  mobile proc void foo(int x, int y, int z) {
2:      int a;
3:      B1
4:      while (B2) {
5:          int q;
6:          B3
7:          suspend;
8:          int w;
9:          B4
10:     }
11:     B5
12: }
```

Figure 2: Sample ProcessJ code with a single interface.

3. Polymorphic Resumption Interfaces

ProcessJ supports polymorphic resumption interfaces [7]. That means, a process can be started with an interface A, execute, suspend, and later resume with a potentially different interface B. This is useful in combination with resources that are only available in the current computational environment, and when the use of ‘dummy’ parameters would otherwise be necessary.

Polymorphic resumption interfaces allows the compiler to do static scope checking using interfaces that would otherwise require unused ‘dummy’ parameters by splitting a single interface into multiple interfaces. Consider Figure 2. If we were to use a single interface, but semantically we only use x , and y in block B_1 and only use w in block B_4 then we are expecting the caller to use two separate *implicit* interfaces while starting *foo*. It is their responsibility to know which variables are actually necessary.

Consider another situation where a process has two interfaces: the first is the reading end of a channel and the second is the reading end of a channel and the writing end of a channel. To implement this process with a single interface, the interface would have to be the super-set of the two interfaces: two channel reading ends and one writing end. There would be no clear distinction from the single interface when each is channel end is valid for use. For instance, if the first channel end is meant only for initialization, and the second and third are meant to be used as the process is passed around. In this case, you would first invoke the process with a valid first channel and with ‘dummy’ channels for the second and third. After the process suspends, you would then resume the process with a ‘dummy’ first channel end and valid second and third ends. What happens when the two are somehow transposed? In the best case the system becomes deadlocked and in the worst case the system is still able to communicate but acts in an unexpected manner. It is exactly this situation we are trying to avoid.

To implement polymorphic resumption interfaces, we use a single variadic function called *run*. A client calls *run* with the appropriate parameters for the current interface of

```

public class AbstractCSProcess implements CSProcess,
Serializable {
    protected int control(int level) {...}
    protected <T> T getParameter(int index) {...}
    protected boolean isRunning(){...}
    protected void resume(Class<?>... parameter-
Types){...}
    public void run(Object... args) {...}
    protected abstract void start();
    protected void suspend(int... targets) {...}
}

```

Figure 3: Methods in AbstractCSProcess

the process. When a process resumes, it checks that the invoked interface and the current resumption interface are compatible.

In the translated Java code we use exceptions, which do not exist in ProcessJ, to indicate when an incorrect interface was used. During run time, the process checks that it is in the correct state for the current interface. The process throws an *IncorrectInterfaceException* during run time if it is not in the correct state. This exception is a *RuntimeException* so it need not be checked. The benefit of throwing the *IncorrectInterfaceException* during run time is it allows a developer to know the interface expected, the interface sent, and the caller of the process when a programming error was made.

4. Control Flow Rewriting

The general outline for ProcessJ code with a suspend is depicted in Figure 1 and the corresponding generated code is depicted in Figure 4. The example is a simple while statement that goes over the basic technique used to rewrite from ProcessJ to Java. Later in the section, we will give examples of how the process changes slightly with each of the other control structures.

Starting from the top and working our way down the code in Figure 4, we will explain each of the rewrites as they appear. The process is first converted into a Java class that extends *AbstractCSProcess*. The *AbstractCSProcess* is a base class that maintains state information and helper methods like *suspend*, *resume*, and *control* for navigating the control structure. The API is displayed in Figure 3

As mentioned previously, all mobile processes are serializable. Serializable processes will allow ProcessJ to send processes in a distributed environment when a distributed run time is available.

The next rewrite converts local variable definitions into fields. This removes the necessity of storing local state at suspend time and restoring it during a resume because everything is stored in fields. To avoid naming conflicts of variables in different scopes a compiler generated prefix is

```

public class foo extends AbstractCSProcess {
    int $b1$a; // original a
    int $b3$q; // original q
    int $b4$w; // original w
    boolean $c1; // original B2

    @Override
    protected void start() {
        // interface 0 (int x, int y)
        int $i0$x, $i0$y;
        // interface 1 (int z)
        int $i1$z;

        switch (control(0)) {
        case 0:
            // interface 0 (int x, int y)
            resume(Integer.class, Integer.class);
            $i0$x = getParameter(0);
            $i0$y = getParameter(1);
            B1
        case 1:
            if (isRunning()) {
                $c1 = B2;
            }
            $1: while ($c1) {
                switch (control(1)) {
                case 0:
                    B3
                    suspend(1,1);
                    return;
                case 1:
                    //interface 1 (int z)
                    resume(Integer.class);
                    $i1$z = getParameter(0);
                    B4
                    $c1 = B2;
                } // end switch
            } // end while
            B5
        } // end switch
    } // end method start
} // end class foo

```

Figure 4: General outline for generated Java code.

generated for each variable. For example, *a* was defined in block *B₁* so it is renamed *\$b1\$a*.

At each control structure that uses an expression, a field is created to store the resulting value. In this case, *B₂* is stored in *\$c1*. Since an expression is only expected to be evaluated once, it cannot be re-evaluated during resumption. To address this, before the expression would normally be evaluated, the evaluated value is stored in a field corresponding to that control point.

```

...
$c1 = B2;
continue $1;
...

```

Figure 5: Example of a continue statement.

The *start* method represents the body of the process. It begins by declaring each of the parameter variables. Again, to avoid naming conflicts the compiler generates prefixes for each interface. For example, *x* is in interface 0 so it is renamed *\$i0\$x*. Next is the base switch statement of the body which splits *B₁* and the *while* statement.

Inside case 0, we resume with a list of classes defined in *interface 0*. The *resume* method checks the provided interface is the same as the expected interface then sets the current state to *RUNNING* and resets the resume target. After that, we need to set the value of each parameter and execute *B₁*.

Java allows fall-through in switch statements. We use fall-through to split up the blocks without breaking the natural control flow of the original program. After block *B₁* executes, block *B₂* needs to be evaluated but only if the process is currently running. If the process was suspended, the expression was already evaluated and stored in *\$c1*.

Now we have reached the second level of control structure and *B₃* is executed. The *suspend* method is then called with a control flow map to the next point of resumption. By 'control flow map', we mean a list of integers that describe the case statements that are selected to bring the process back to the next resumption point.

The process stores the control flow map and saves its state as *SUSPENDED* and returns control to the caller. The caller now has a reference to an object that can be serialized and saved, or sent over a channel, or immediately called again with the next interface.

When the process is resumed, the *control* method is called. The *control* method looks at the control flow map saved by the last suspend. It then jumps to case 1 and evaluates the previously stored value *\$c1* at *while*. The *control* method then looks up the next level of control in the control flow map and again jumps to case 1.

The *resume* method checks the interface for the new *interface 1* and sets the state to *RUNNING*. The process then sets the value of the parameters and executes *B₄*. The expression *B₂* is then re-evaluated and stored in *\$c1* and the process continues to loop.

In the subsequent subsections we describe special circumstances for each of the Java control flow structures. In each of the examples you can replace the while statement of Figure 4 with the given generated code.

4.1 break

The *break* statement poses a small problem. In Java, *break* is used to terminate the enclosing *for*, *while*, *do-while* loop

```

...
if (B2) {
    B3
    suspend resume with (int z);
    B4
} else {
    ...
}
...

```

Figure 6: Example if statement in ProcessJ.

```

...
if (isRunning()) {
    $c1 = B2;
}
if ($c1) {
    switch(control(1)) {
    case 0:
        B3
        suspend(1,1);
        return;
    case 1:
        //interface 1
        resume(Integer.class);
        $i1$z = getParameter(0);
        B4
    }
} else {
    ...
}
...

```

Figure 7: Example of if statement with suspend.

or *switch* statement. We are using *switch* statements to jump through blocks to resume points. If an unlabeled *break* is used in a block it would break out of the control flow *switch* instead of the original intended structure. We get around this by adding a label to each control structure and labeling unlabeled *break* statements.

4.2 continue

The *continue* statement skips the rest of the current iteration in a looping construct. Since a continue will skip updating the stored expression, the looping expression needs to be re-evaluated before a *continue* is executed. A *continue* is then rewritten as in Figure 5.

4.3 if else

The *if* statement needs almost no special treatment from that described above. An example of an *if* statement rewrite can be seen in Figure 7. The expression *\$c1*, is stored as a boolean. The use of *\$c1* ensures the expression is only evaluated while the process is in a *RUNNING* state.

There is no need to specify a label for the *if* statement. A *break* statement can be used to escape an *if* but it needs to specify a label to do so.

4.4 switch

The *switch* statement is rewritten similar to an *if* statement as seen in 9. The expression can be of type *byte*, *short*, *char*, and *int* [8], so the stored expression needs to be of the evaluated type.

Since *switch* statements allow fall-through from case to case, we need to consider the situation where B_2 evaluates to case 0, there is no *break* statement in case 0 and the control falls through to case 1. In this situation, we need to update the stored value $\$c1$ at each case. Updating the stored value allows the process to resume to the last case statement the process was in.

Unlike the *if* statement, a *switch* needs a label. If an unlabeled *break* is within the *switch* statement, the *break* would inadvertently escape the generated control flow *switch* statement instead of the intended *switch*.

4.5 do-while

Figure 11 shows a *do-while* statement. Like other looping constructs and *switch*, *do-while* requires a label for *break* statements. Unlike other control structures, it is not necessary to store the result of the expression. Because the expression is evaluated at the end of the loop, it is never evaluated in the resumption process.

4.6 while

A *while* loop, was used in the main example in Figure 4. At the end of the loop, the expression is re-evaluated and stored. There is no need to check if the process is currently running at the end of a *while* because a suspended process will never reach this code.

The *while* statement also requires a label for *break* statements.

```

...
$1:switch ( $B_2$ ) {
  case 0:
     $B_3$ 
    suspend resume with (int z);
     $B_4$ 
  case 1:
    ...
  default:
    ...
}
...

```

Figure 8: Example of switch statement with suspend in ProcessJ.

4.7 for

A *for* statement is best broken down into a *while* loop as seen in Figure 13. Before the loop, the expression and the initial value are both set only if the process is in *RUNNING*. There is a label for *break* statements. At the end of the loop, the update is executed and the conditional expression is re-evaluated and stored.

Similar to the *while* statement, there is no need to check if the process is running because a suspended process will never reach this code.

4.8 Process

When a mobile process contains another mobile process, each process maintains its own state. For instance, if you invoke a mobile process B , from within a process A , and B suspends execution, that will only return control to A . The A process may decide to resume process B , or it may pass it down a channel.

It would also be possible for A to suspend and continue to hold a reference to process B . When A resumes it would also be possible to resume process B from A without any extra work other than what has already been explained in this paper.

```

...
if (isRunning()) {
   $\$c1 = B_2$ ;
}
$1:switch( $\$c1$ ) {
  case 0:
     $\$c1 = 0$ ;
    switch (control(1)) {
      case 0:
         $B_3$ 
        suspend(1,1);
        return;
      case 1:
        // interface 1
        resume(Integer.class);
         $\$i\$z = getParameter(0)$ ;
         $B_4$ 
    }
  case 1:
     $\$c1 = 1$ ;
    ...
  default:
     $\$c1 = 2$ ; // not in other cases
    ...
}
...

```

Figure 9: Example of switch statement with suspend.

```

...
$1: do {
    B3
    suspend resume with (int z);
    B4
} while(B2);
...

```

Figure 10: Example of do-while loop with suspend in ProcessJ.

```

...
$1: do {
    switch(control(1)) {
    case 0:
        B3
        suspend(1,1);
        return;
    case 1:
        //interface 1
        resume(Integer.class);
        $i$z = getParameter(0);
        B4
    }
} while(B2);
...

```

Figure 11: Example of do-while loop with suspend.

```

...
for(init_expression; B2; update_expression) {
    B3
    suspend resume with (int z);
    B4
}
...

```

Figure 12: Example of for loop with suspend in ProcessJ.

5. Related Work

Mobile processes can be seen as a form of *process continuation* [9]. Unlike a traditional continuation, a process continuation represents the rest of a sub-computation from a given point in that sub-computation. Each process in ProcessJ executes in its own execution context. Therefore, a continuation of that process represents the control state of that one process, not the system as a whole.

An implementation of non-transparent weak mobility is available in Java through the use of `jbsp.mobile` [10], [11]. Though this implementation does allow process mobility, the end programmer needs to save all state and there is no way to save control state. One benefit to using `jbsp.mobile` is that it manages the class loading while communicating across JVMs.

Since ProcessJ already manages the state of the mobile process, `jbsp.mobile` may eventually find a place in the distributed run time. ProcessJ transparent strong mobility

```

...
if (isRunning()) {
    init_expression;
    $c1 = B2;
}
$1: while($c1) {
    switch(control(1)) {
    case 0:
        B3
        suspend(1,1);
        return;
    case 1:
        //interface 1
        resume(Integer.class);
        $i$z = getParameter(0);
        B4
    }
    update_expression;
    $c1 = B2;
}
...

```

Figure 13: Example of for loop with suspend.

and `jbsp.mobile`'s ability to manage dynamic class loading would be a strong combination.

Stefan Fünfroeken describes in [12] how to transparently migrate the state of a thread in Java. The approach described uses a preprocessor to instrument the Java code so no bytecode rewriting was necessary.

The difference between Fünfroeken's approach and ours is the need to migrate the entire thread stack. In ProcessJ, we are only interested in restoring the state of a single process, not the thread stack. Also, since we are able to convert all locals variables to fields, we need not save any state other than the current control state. This makes our approach much simpler for the implementation of ProcessJ.

Pedersen and Kauke describe in [1] how to provide transparent mobility in the JVM using a combination of code generation and bytecode rewriting. Since this paper is in large part an improvement on this work, let's look at how our implementation differs in greater detail. Our implementation is accomplished without bytecode rewriting, we use the new concept of polymorphic resumption interfaces, and there is no need to save local state before a suspend.

The bytecode rewriting adds an extra step to the flow. After Java code is produced, it must be compiled, then the bytecode rewritten before it can execute. In our approach, we produce Java code that is ready to compile and execute without modification.

In this approach we allow for polymorphic resumption interfaces where as [1] implements "resumability with parameter changes". Polymorphic resumption interfaces are a step above this because not only are the parameters allowed

to change, but the interfaces changes as well.

One other difference lies in the points just before suspension and resumption. In [1] all state is stored in an activation record. The activation record is implemented as an array of objects on the process. Before a suspend, all the local state is saved into the activation record, and on resumption all local state is restored back to the proper local variables. Our implementation simplifies this drastically by moving all local variables into fields so there is no need to store values during suspension and restore during resumption.

6. Conclusion

In this paper, we have shown how the ProcessJ compiler can provide transparent process mobility using only code generation. We have also shown how to implement polymorphic resumption interfaces and describe the rewriting steps required to provide these features in ProcessJ.

7. Future Work

It is still necessary to perform the static scope checking proposed in [7]. This allows developers to know exactly where parameters are can be referenced.

All the necessary future work mentioned in [1] is still relevant. Handling channels inside mobile processes and other local resources still need to be resolved though the polymorphic resumption interfaces should help. Channels and other local resources in mobile processes could be handled by only allowing them in parameters and not stored as local variables. As mentioned previously, it would also be nice to use jscsp.mobile's dynamic class loading to load mobile processes across JVM.

To simplify matters a little, the code demonstrated is not yet integrated into JCSP. However, with little effort in the base class it should be possible to execute these processes as a *CSPProcess*.

References

- [1] J. B. Pedersen and B. Kauke, "Resumable Java Bytecode - Process Mobility for the JVM," in *The thirty-second Communicating Process Architectures Conference, CPA 2009, organised under the auspices of WoTUG, Eindhoven, The Netherlands, 1-6 November 2009*, 2009, pp. 159–172.
- [2] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, vol. 21, pp. 666–677, August 1978.
- [3] R. Milner, *Communicating and mobile systems the pi-calculus*. Cambridge[England] ;;New York: Cambridge University Press, 1999.
- [4] P. Welch and F. Barnes, "Communicating Mobile Processes: introducing *occam-π*," in *25 Years of CSP*, ser. Lecture Notes in Computer Science, A. Abdallah, C. Jones, and J. Sanders, Eds., vol. 3525. Springer Verlag, April 2005, pp. 175–210.
- [5] G. Cugola, C. Ghezzi, G. Picco, and G. Vigna, "Analyzing mobile code languages," in *Mobile Object Systems Towards the Programmable Internet*, ser. Lecture Notes in Computer Science, J. Vitek and C. Tschudin, Eds. Springer Berlin / Heidelberg, 1997, vol. 1222, pp. 91–109.
- [6] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten, "Portable support for transparent thread migration in java," in *Agent Systems, Mobile Agents, and Applications*, ser. Lecture Notes in Computer Science, D. Kotz and F. Mattern, Eds. Springer Berlin / Heidelberg, 2000, vol. 1882, pp. 377–426.
- [7] J. B. Pedersen and M. Sowders, "Static Scoping and Name Resolution for Mobile Processes with Varying Resumption Interfaces," in *Need the book title*, 2011.
- [8] K. Arnold, J. Gosling, and D. Holmes, *Java(TM) Programming Language, The (4th Edition)*. Addison-Wesley Professional, 2005.
- [9] R. Hieb and R. K. Dybvig, "Continuations and concurrency," in *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, ser. PPOPP '90. New York, NY, USA: ACM, 1990, pp. 128–136. [Online]. Available: <http://doi.acm.org/10.1145/99163.99178>
- [10] K. Chalmers and J. M. Kerridge, "jscsp.mobile: A Package Enabling Mobile Processes and Channels," in *Communicating Process Architectures 2005*, sep 2005.
- [11] K. Chalmers, J. M. Kerridge, and I. Romdhani, "Mobility in JCSP: New Mobile Channel and Mobile Process Models," in *Communicating Process Architectures 2007*, A. A. McEwan, W. Ifill, and P. H. Welch, Eds., jul 2007, pp. 163–182.
- [12] S. Fünfroeken, "Transparent migration of java-based mobile agents," in *Mobile Agents*, ser. Lecture Notes in Computer Science, K. Rothermel and F. Hohl, Eds. Springer Berlin / Heidelberg, 1998, vol. 1477, pp. 26–37.

Supporting Ordered Multiprefix Operations in Emulated Shared Memory CMPs

Martti Forsell

Platform Architectures Team
VTT
Oulu, Finland

Jussi Roivainen

Digital Systems Design Team
VTT
Oulu, Finland

Abstract - Shared memory emulation is a promising technique to address programmability and performance scalability concerns of chip multiprocessors (CMP) because it provides implied synchrony in execution of machine instructions, efficient latency hiding technique, and enough effective bandwidth to route all the memory references even with the heaviest workloads. In our earlier research we have proposed an architectural solution to support concurrent memory access and multioperations on emulated shared memory CMPs with a help of active memory units attached to memory modules. While this solution provides faster memory access than other known solutions with minor silicon area and power consumption overheads, the results of multiprefixes are unfortunately not in order forcing one to use a relatively slow logarithmic multiprefix algorithm for ordered multiprefixes. In this paper we propose an architectural technique for supporting a limited number of concurrent ordered multiprefix operations in emulated shared memory CMPs. The solution is based on adding special multiprefix arrays to active memory units. Performance, silicon area, and power consumption evaluations are given.

Keywords: Parallel computing, CMP, ordered multiprefix, concurrent memory access, computer architecture

1 Introduction

Shared memory emulation [Ranade91] is a promising technique to address programmability and performance scalability concerns of *chip multiprocessors* (CMP). This is because it provides implied synchrony in execution of machine instructions, efficient latency hiding technique, and enough effective bandwidth to route all the memory references even with the heaviest random and concurrent access workloads. Synchronous execution is considered to make programming easier because a programmer does not need to synchronize threads of execution explicitly after each global memory access but can rely on the hardware to take care of that automatically. Latency hiding used in shared memory emulation makes use of the high-throughput computing scheme [Beck97], where other threads are executed while a thread refers to the global shared memory. Since the throughput computing scheme employs parallel slackness extracted from available thread-level parallelism, it is considered to provide remarkably better scalability than traditional symmetric multiprocessors and non uniform memory access systems relying on snooping or directory-based cache coherence mechanisms and therefore suffering from limited bandwidth or directory access delays and heavy coherence traffic.

We have proposed an architectural solution to support concurrent memory access and multioperations on emulated shared memory CMPs with a help of active memory units attached to

memory modules [Forsell06a]. While the solution indeed provides faster memory access than other known solutions with minor silicon area and power consumption overheads, the results of multiprefixes are unfortunately not in order forcing one to use a relatively slow logarithmic multiprefix algorithm for ordered multiprefixes. In this paper we propose an architectural technique for supporting a limited number of concurrent ordered multiprefix operations in emulated shared memory CMPs. The solution is based on adding special *multiprefix arrays* to active memory units. Performance, silicon area, and power consumption evaluations are given.

1.1 Related work

Architectures for shared memory emulation, known also as *emulated shared memory* (ESM) architectures, have been studied from the 70's when the ideal shared memory machine, the *parallel random access machine* (PRAM) [Fortune78] was invented: Schwartz proposed ultracomputers with network switches to combine requests destined for the same memory location [Schwartz80]. Ranade outlined a method to emulate PRAM-like shared memory [Ranade91]. Forsell outlined a scalable on-chip computing architecture with efficient instruction-level parallelism exploitation for general purpose parallel computers employing the PRAM model [Forsell02]. The idea of partial and limited concurrent memory access for synchronous CMPs was presented in [Forsell05]. It takes, however, as many as three steps to make a full concurrent access and provides only a low number of memory locations for which concurrent access is allowed. The idea was further extended to full concurrent memory access and multioperation support with the help of step caches and scratchpads in [Forsell06], but the technique does not preserve the ordering of multiprefixes. Vishkin introduced the explicit multithreaded architecture including a multiprefix computation unit for realizing PRAM-like computing but the used synchronization scheme is more relaxed than in strict PRAM limiting its applicability [Vishkin11].

The rest of the article is organized so that in Section 2 we describe the emulated shared memory system, the novel architectural technique supporting ordered multiprefix operations is proposed in Section 3, in Section 4 we evaluate the proposed technique on our emulated shared memory CMP framework and give rough silicon area and power consumption estimations, and finally in Section 5 we give our conclusions.

2. Shared memory emulation

The main idea in shared memory emulation is to provide a user an illusion of ideal shared memory although the underlying

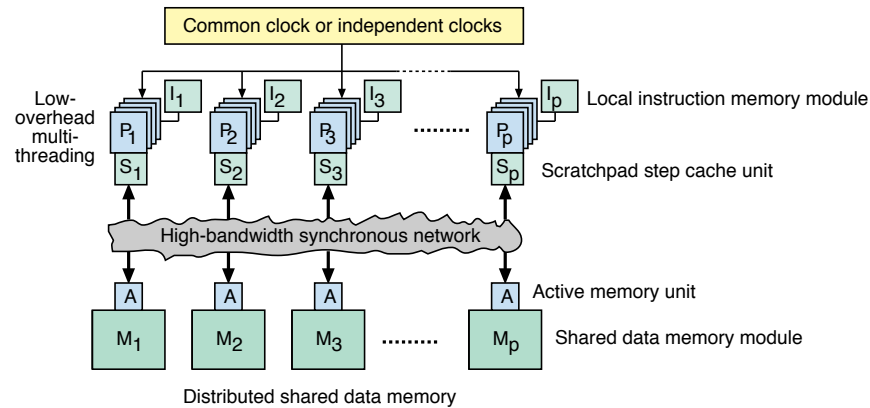


Figure 1. Emulated shared memory system supporting concurrent memory access, multioperations and arbitrarily ordered multiprefixes.

architecture has a physically distributed memory. The properties of ideal shared memory are best captured by the PRAM model, which abstract away asynchronicity in execution of threads and non-uniformities (latency and need for partitioning) of memory access. PRAM lets a programmer to focus on intrinsic parallelism of the computational problem and parallel algorithm design instead of being forced to orchestrate asynchronous, possibly non-uniform and implementation-dependent low-level issues. Unfortunately the direct implementation of an ideal, PRAM-style shared memory has proved to be physically infeasible with current silicon technology if the number of processors is say higher than, say 4 [Forsell94]. This is because the wiring area (and the power dissipation) of multiport memory chip raises quadratically as the number of processors increases with respect to a single ported memory of the same capacity. In this section, we describe the main principles and architectural techniques of shared memory emulation, introduce existing solutions for implementing concurrent memory access and multiprefix operations.

2.1 Principles

A typical scalable architecture to emulate shared memory on a silicon platform consists of a set of processor cores connected to a distributed shared memory via a physically scalable high-bandwidth interconnection network (see Figure 1). The main idea is to provide each processor core with a set of threads that are executed efficiently in an interleaved manner and hide the latency of the network. As a thread makes a memory reference, the thread is changed and second thread can make its memory request, and so on. No memory delay will occur assuming the reply of the memory reference of the thread arrives to the processor core before the thread is put back to execution [Ranade91]. This requires that the bandwidth of the network is high enough and hot spots can be avoided in pipelined memory access traffic. Synchronicity between consecutive instructions can be guaranteed by using an elastic synchronization wave between the steps [Leppänen96].

2.2 Implementation techniques

In order to efficiently emulate shared memory on a top of a distributed memory system, processors need to be multithreaded [Valiant90, Leppänen96]. Such a multithreading can be imple-

mented as a T_p -stage, cyclic, in-order (interleaved) interthread pipeline, which provides hazard-free execution for hiding the latency of the memory system, maximizing overlapping of the execution of threads, and minimizing the register access delay. Switching between threads does not slow down operation of the processor, because threads proceed in the pipeline only during the forward time. If a thread tries to refer memory when the network is busy, the pipeline is suspended until the network becomes available again. After issuing a memory read, the thread can wait the reply for at most $M_w < T_p$ clock cycles before the pipeline freezes until the reply arrives. A processor is composed of F functional units, a hash address calculation unit, and T_p sets of R registers. The scheduling of operations is static since dynamic techniques might conflict with the synchronous thread level parallel (TLP) execution. The PRAM model is linked to the architecture so that a full cycle in the pipeline corresponds typically to a single PRAM step. During a step, each thread of each processor of the CMP executes an instruction including at most one shared memory reference subinstruction. Therefore a step lasts for multiple, at least T_p , clock cycles.

There are two types of memory modules, data memory modules and instruction memory modules, that are accessed via the data and instruction memory ports of processors, respectively (see Figure 1). All the data is located to physically distributed but logically shared data memory modules emulating the ideal PRAM memory. Instruction memory modules are aimed to keep the program code for each processor. The data and instruction memory modules of size S_{sd} and S_i bytes, respectively, are isolated from each other to guarantee parallel high-bandwidth data and instruction streams to processors.

The communication network connects processors to distributed memory modules so that sufficient throughput and low enough latency can be achieved for random communication patterns with a high probability as outlined in [Ranade91, Leppänen96]. Suitable scalable intercommunication topologies include sparse or under populated networks, e.g. variants of two-dimensional meshes providing fixed degree nodes as well as fixed length of interconnection lines independently on the number of processors. To maximize the throughput for read-intensive portions of code, one can use separate lines for references going from processors to memories and for replies from memories to processors. Memory locations are distributed across the data modules by a randomly chosen polynomial

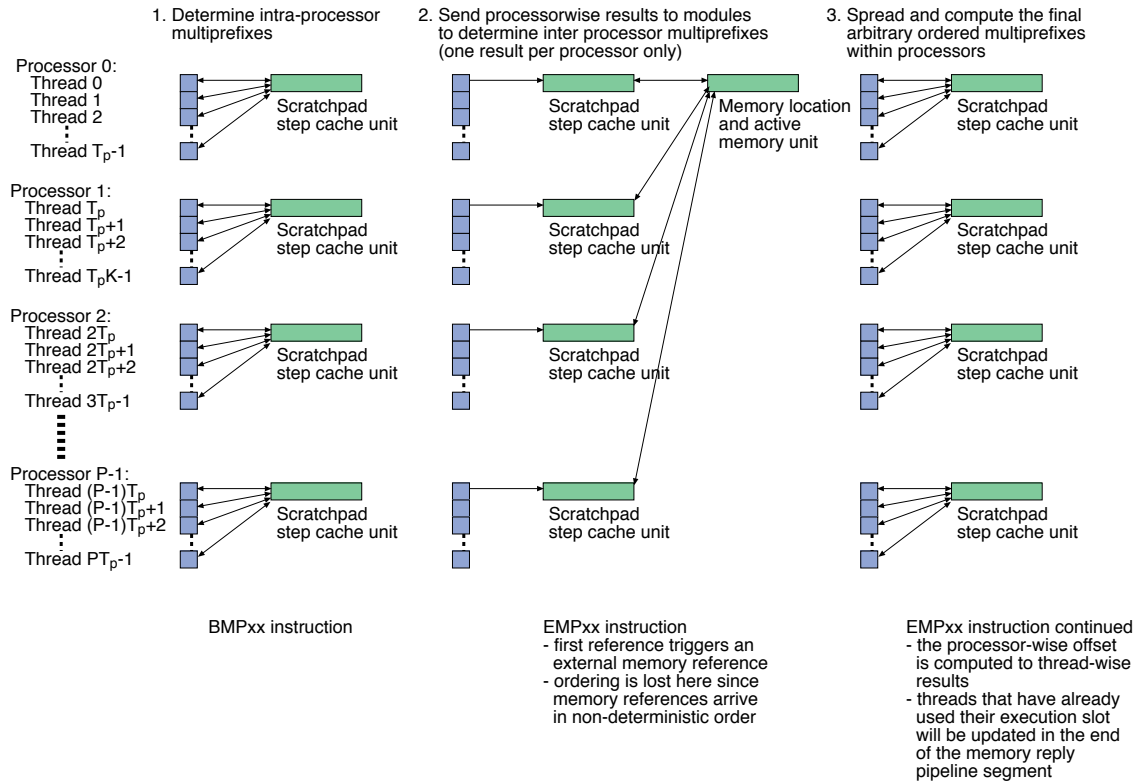


Figure 2. Multiprefix using the two-level approach that does not preserve the ordering of multiprefix operations.

hashing function for avoiding congestion of messages and hot spots [Ranade91, Dietzfelbinger94].

2.3 Concurrent memory access and multiprefix operations

Concurrent reads and writes to memory locations can be implemented using step caches. For a concurrent read, all threads participating the access give the same results. In the case of a concurrent write, the data of an arbitrary thread participating the write will be written to the target location. Step caches are associative memory buffers in which data stays valid only to the end of ongoing step of multithreaded execution [Forsell05]. The main contribution of step caches to concurrent accesses is that they step-wisely filter out everything but the first reference for each referenced memory location. This reduces the number of requests per location to P allowing them to be processed sequentially on a single ported memory module assuming $T_p \geq P$. Step caches operate similarly as ordinary caches with a few notable exceptions: Each time a multithreaded processor refers to the shared data memory a step cache search is performed.

Scratchpads are addressable memory buffers that are used to store memory access data to keep the associativity of step caches limited in implementing multioperations and thread bunches with a help of step caches, and minimal on-core and off-core ALUs that take care of actual intra-processor and inter-processor computation for multioperations [Forsell06] (see Figure 1). Scratchpads are coupled with step caches into so called scratchpad step cache units. A scratchpad step cache unit consists of a T_p -line scratchpad, a T_p -line step cache, and a simple multioperation ALU for executing incoming concurrent ref-

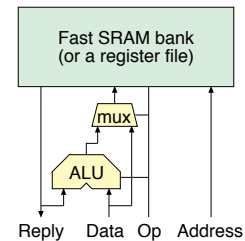


Figure 3. Active memory unit.

ferences, multioperations and arbitrary ordered multiprefixes sequentially.

Multioperations can be implemented as two consecutive single step operations. During the first step, a starting operation (BMPxx for arbitrary ordered multiprefix operations) executes a processor-wise multioperation against a step cache location without making any reference to the external memory system (see Figure 2). During the second step, an ending operation (EMPxx for arbitrary ordered multiprefix operations) performs the rest of the multioperation so that the first reference to a previously initialized memory location triggers an external memory reference using the processor-wise multioperation result as an operand. The external memory references that are targeted to the same location are processed in the active memory units of the corresponding memory module according to the type of the multioperation. An active memory unit consists of a simple ALU and fetcher (see Figure 3). In the case of arbitrary ordered multiprefixes the reply data is sent back to scratchpads of participating processors. The consecutive references are completed by applying the reply data against the step cached reply data.

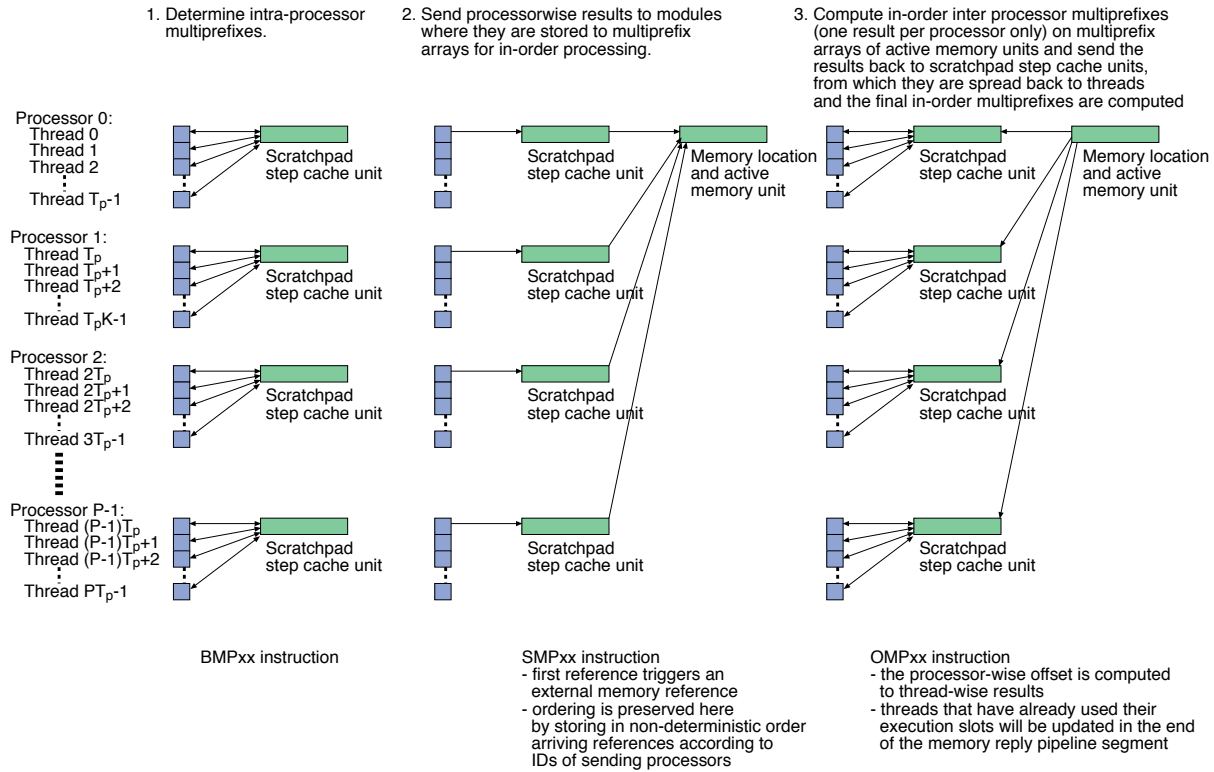


Figure 4. Multiprefix using the proposed three-level multiprefix array technique that preserves the ordering of multiprefix operations.

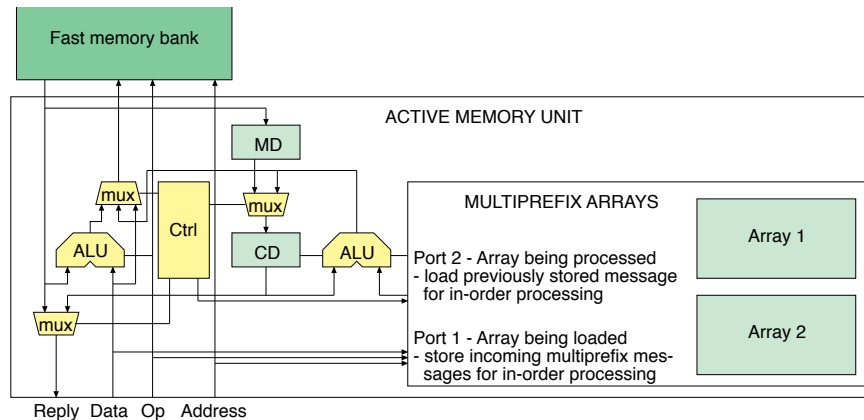


Figure 5. Active memory unit for the proposed in-order multiprefix solution.

3. Ordered multioperations

The fastest known implementation of concurrent memory access and arbitrary ordered multiprefixes—called here as the *baseline solution*—fails to preserve the ordering of participating threads in multiprefixes and therefore introduces a logarithmic slowdown for ordered multiprefixes [Forsell06a]. This is because memory references arrive at the destination module in the order dependent on the distance of target memory module from the source processors, traffic situation, and relative timing of threads compared to each others. In order to retain the high performance provided by this baseline solution in concurrent memory access and to restore the ordering of references participating to multiprefix operations for an arbitrary number of threads, we propose using a three step algorithm named here as the *multi-*

prefix array technique, operating identically to the baseline solution during step one, adding an array to the active memory unit for storing and ordering references according to their source processor IDs during step two, and control logic for *processing* the references in order during step three (see Figure 4). To allow for an overlapped processing of this kind of dual instruction solution, an additional array and storage for keeping the target memory value are needed so that one is dedicated for processing (processing array, cumulative data register) while another is filled with references (loading array, data register). Thus, the modified active memory units consist of an ALU and a multiplexer like the baseline active memory unit but adds also two multiplexers, control logic, another ALU, two registers, and two multiprefix arrays (see Figure 5).

<pre> #include "e+.h" // Baseline version, T=O(log N) in e #define size 32768 int source_[size]; int main() { int i; for_(i=1, i<_number_of_threads, i<=1, if (_thread_id-i>=0) source_[_thread_id] += source_[_thread_id-i];); } </pre>	<pre> #include "e+.h" // Proposed version, T=O(1) in e #define size 32768 int sum_=0; int source_[size]; int main() { int p; prefix(p,MPADD,&sum_,1); source_[_thread_id]=p; } </pre>
<pre> ; Proposed version in MBTAC assembler ; R1 = address of _thread_id, _thread_id, _thread_id<<2 ; R2 = Prefix intermediate result, prefix final result ; R3 = address of _source_ OPO _sum_ OP1 1 BMPADD0 O1,00 WB2 M0 ; Step 1 OPO _sum_ SMPADD0 R2,00 ; Step 2 OPO _sum_ OP1 _thread_id ADD0 O1,R32 OMPADD0 R2,00 WB1 A0 WB2 M0 ; Step 3 OPO _source_ LD0 R1 WB3 O0 WB1 M0 OPO 2 SHL0 R1,00 WB1 A0 ADD0 R1,R3 STO R2,A0 </pre>	

Figure 6. Ordered multiprefix add as e (for the baseline and proposed solutions) and assembler programs (for the proposed solution).

The modified active memory unit works like the baseline unit for loads and stores. For multiprefixes it stores the incoming references that arrive in a semi-random order to the loading multiprefix array addressed by the referencing processor ID, sets the corresponding in-use bit of the element of the array, and fetches the target memory location value to register MD (SMPxx instructions). During the next step, the control logic first switches the arrays so that current processing array becomes loading array and vice versa, clears the in-use bits of the new loading array in parallel, advances the memory data from register MD to register CD, processes the now ordered references against this data skipping the unused elements and sending the old values of register CD back to issuing processors running OMPxx instructions, stores the results back to register CD, and finally stores the final result back to the memory.

The proposed multiprefix array solution allows for a single multiprefix computation per a memory module (or processor) per step but two overlapping multiprefixes can share the same module—if they are targeted to the very same memory location, the control logic stores the final value also to register MD. While this may sound quite limited amount of concurrent prefixes, it should be noted that the best speedups are gained in cases in which as many threads as possible are participating in a multiprefix, the extreme case reducing only to single multiprefix in which all the threads are participating.

This kind of active memory units can be used to realize the full *multiprefix concurrent read concurrent write* (MCRCW) PRAM model. A potential problem with multioperations in the baseline solution is that active memory units in practice need to both read and write accesses memory locations for each participating thread. This can easily lead to limiting the number of memory locations available for multioperations, adding fast caches to memory modules, or doubling the speed of the memory. The proposed solution eliminates these problems for ordered multiprefixes by reading the memory only once per incoming reference, and storing the result only once at the end of multiprefix computation. Also the arrays are single-ported decreasing the silicon area needed for them and making the power consumption potentially modest.

From a point of view of programming, using new fast ordered multiprefix operations of an MCRCW-enabled ESM is simple: A programmer needs just to apply MCRCW-aware primitives to get up to a logarithmic performance boost. Figure 6

shows a logarithmic prefix algorithm for the baseline solution and a constant time prefix employing the proposed solution for full MCRCW. It shows the baseline and MCRCW versions of the prefix benchmark as e-programs and the MCRCW version in our CMP framework assembler program.

4. Evaluation

In order to evaluate the performance and estimate the silicon area, and power consumption of the proposed multiprefix array technique, we applied it to the ECLIPSE CMP framework being developed at VTT [Forsell02, Forsell10].

4.1 Preliminary performance simulations

We measured the performance of the multiprefix solutions by simulating execution of five prefix problems that can be used as a primitive of parallel computing (see Table 1) in six CMP configurations having 4 to 64 512-threaded MBTAC processor cores (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 modified for the proposed solution. The results of the simulations are shown in Figure 7.

In order to roughly compare the goodness of proposed solution to the alternative solutions, we determined the number of steps needed for exclusive and concurrent memory accesses, associative multioperations and both arbitrary ordered and fully ordered multiprefixes in the baseline solution, the proposed solution, and combining solution guaranteeing ordering but introducing always the sorting phase (see Table 3).

4.2 Silicon area and power consumption

We estimated the silicon area, power consumption, and maximum clock frequency of unoptimized ESM CMPs with 4, 16 and 64 processors using the recently proposed performance-area-power model of ESM CMPs [Forsell08] assuming high-performance 65 nm silicon technology, minimum global wiring pitch for interconnects, 1 MB data SRAM and 8 kiloinstructions totally uncompressed program SRAM per processor. The model features over 100 parameters and determines the number of gates

Benchmark	T_{base}	P_{base}	W_{base}	T_{prop}	P_{prop}	W_{prop}	N	Description
aprefix	1	N	N	N	1	1	T	Arbitrary ordered multiprefix of a table of N integers
prefix-x	$\log N$	N	$N \log N$	N	1	1	T	Ordered multiprefix of a table of N/x integers ($x=1, 4, 16, 64$ concurrent multiprefix computations)

Table 1. Benchmarks used in evaluation.

Configuration	E4	E16	E64	C4	C16	C64
Multiprefix processing machinery	baseline	baseline	baseline	proposed	proposed	proposed
Number of processors	4	16	64	4	16	64
Number of functional units	P	4	4	4	4	4
Number of threads per processor	T_p	512	512	512	512	512
Total number of threads (in tests)	T	2k	8k	2k	8k	32k
Number of switches	S	4	16	256	4	256
Size of data memory (MB)	S_m	4	16	64	4	64

Table 2. CMP configurations used in the evaluation.

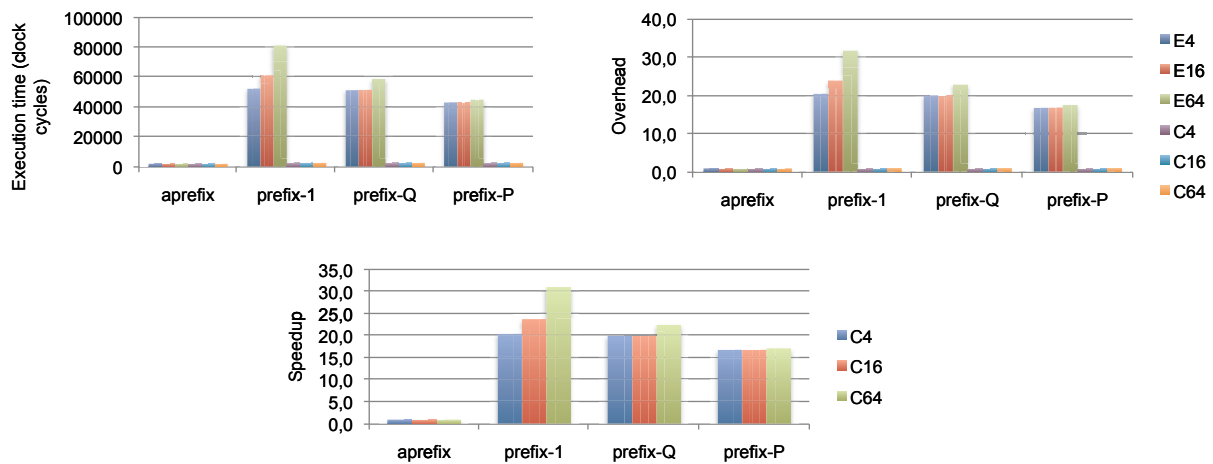


Figure 7. Execution time (top left), overhead with respect to ideal machine with the same instruction set (top right), and achieved speedup

Operation	Combining solution	Baseline solution	Multiprefix arrays
Definition	[Ranade91]	[Forsell06a]	<This paper>
Exclusive load	2-3	1	1
Exclusive store	2-3	1	1
Concurrent load	2-3	1	1
Concurrent store	2-3	1	1
Multiplication ($T_{participating} \leq T^{0.5}$)	2-3	1	1
Arbitrary ordered multiprefix ($T_{pt} \leq T^{0.5}$)	2-3	1	1
Ordered multiprefix ($T_{pt} \leq T^{0.5}$)	2-3	$O(\log T_{pt})$	2 if $N_{cmp} \leq P$, $O(\log T_{pt})$ if $N_{cmp} > P$
Multiplication ($T_{pt} > T^{0.5}$)	2-3	2	2
Arbitrary ordered multiprefix ($T_{pt} > T^{0.5}$)	2-3	2	2
Ordered multiprefix ($T_{pt} > T^{0.5}$)	2-3	$O(\log T_{pt})$	3 if $N_{cmp} \leq P$, $O(\log T_{pt})$ if $N_{cmp} > P$

Table 3. Execution time of memory related operations in the evaluated solutions in steps. Multiplications associative cumulative operations like multiprefixes but no return values are sent back to processors, just the content of the target memory location is altered. (N_{cmp} =number of concurrent multiprefixes, T_{pt} =number of participating threads)

by summing the gate counts of elements together, determines the area by multiplying the gate counts in different categories with typical gate size in that category, assuming typical overhead, and adding the area occupied by interconnect wiring. The clock cycle duration is predicted based on a wire delay estimate obtained using a parasitic capacitance model of parallel wires and the length of interconnect wiring calculated from the dimensions of a processor storage module. The power consumption is determined in the same manner as the silicon area by employing typical dynamical and static power consumption

per gate, taking the predicted clock cycle into account and adding the power consumption of the interconnection network wiring. The results given by the model are shown in Figure 8.

4.3 Discussion

As expected CMPs using the proposed multiprefix array solution executed ordered prefix programs much faster than the baseline CMPs if the number of concurrent multiprefixes does not exceed the number of memory modules (processors). The

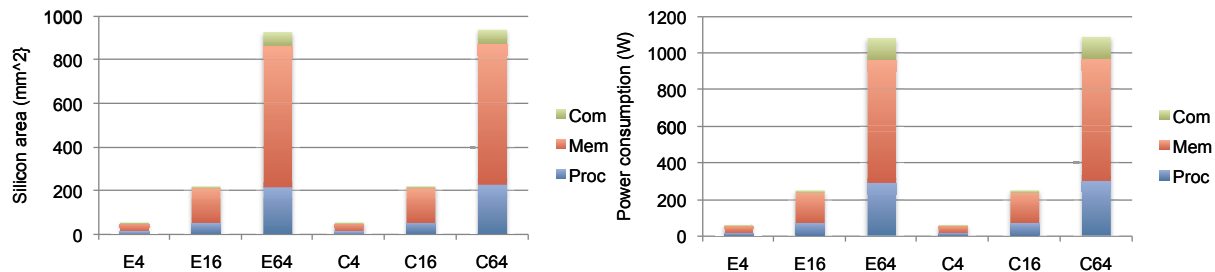


Figure 8. Silicon area (left), and power consumption (right) estimates for non-optimized implementations of CMPs using high-performance 65 nm silicon technology with minimum global wiring pitch for interconnects implying clock frequency of 1.29 GHz assuming 1 MB data SRAM and 8 kiloinstructions program SRAM per processor for determining the clock frequency.

individual speedups ranged from 16.8 to 31.0 while the average speedups were 19.0, 20.0 and 22.8 for C4, C16 and C64, respectively. The results are better than those predicted by the rough logarithmic speedups, ignoring the effect of constant factors and that not every part of the programs is necessarily speed up, 11, 13 and 15, for C4, C16 and C64, respectively. The fact that the overheads of multiprefixes were very small with respect to an ideal PRAM with the similar instruction set, confirms that the efficiency of the proposed technique is high. Finally, the silicon area and power figures as well as maximum clock frequency of an unoptimized design, 220 mm², 250 W at 1.29 GHz for a 16-core CMP, respectively, do not differ radically from those of typical commercial CMPs with similar number of registers while the area and power overheads of the proposed MCRCW implementation with respect to the baseline are less than 1.2%.

5. Conclusions

We have described an architectural technique supporting a limited number of concurrent ordered multiprefix operations in concurrent memory access-aware ESM CMPs. The solution is based on adding special multiprefix arrays to active memory units. According to our evaluations, the technique indeed provides high speedups with respect to the baseline CMPs while keeping the silicon area and power consumption overheads very low. The measured average speedups ranged from 19.0 on a 4-processor chip to 22.8 on a 64-processor chip. While the proposed technique supports up to P ordered simultaneous multiprefix operations, the baseline solution supports P or more simultaneous prefixes as long as the threads participating to a single multiprefix belong to the same processor.

Our future work includes investigating whether more than one concurrent ordered multiprefix per a memory module could be issued, the relatively high buffering requirements could be decreased, and whether there exists a way to make concurrent access even faster than in the proposed solution. Finally, we aim to investigate memory module level caching solutions to make all memory locations equal with respect to multioperations.

Acknowledgements

This work was supported by the REPLICA frontier research project of VTT and grant 128733 of the Academy of Finland.

References

[Beck97] A. Beck, High Throughput Computing: An Interview with Miron Livny, 1997. HPCWire.

[Dietzfelbinger94] M. Dietzfelbinger et. al.: Dynamic Perfect Hashing: Upper and Lower Bounds, SIAM Journal on Computing, Vol. 23, No. 4 1994, pp. 738-761.

[Forsell94] M. Forsell, Are Multiport Memories Physically Feasible?, Computer Architecture News 22, 4 (September 1994), 47-54.

[Forsell97] M. Forsell, MTAC—A Multithreaded VLIW Architecture for PRAM Simulation, Journal of Universal Computer Science 3, 9 (1997), 1037-1055.

[Forsell02] M. Forsell, A Scalable High-Performance Computing Solution for Network on Chips, IEEE Micro 22, 5 (September-October 2002), 46-55.

[Forsell05] M. Forsell, Step Caches—a Novel Approach to Concurrent Memory Access on Shared Memory MP-SOCs, Proc. 23th IEEE NORCHIP, November 21-22, 2005, Oulu, Finland, 74-77.

[Forsell06a] M. Forsell, Realizing Multioperations for Step Cashed MP-SOCs, Proc. SOC'06, November 14-16, 2006, Tampere, Finland.

[Forsell06b] M. Forsell, Reducing the associativity and size of step caches in CRCW operation, In the Proceeding of 8th Workshop on Advances in Parallel and Distributed Computational Models (in conjunction with the 20th IEEE International Parallel and Distributed Processing Symposium, IPDPS 06), April 25, 2006, Rhodes, Greece.

[Forsell08] M. Forsell and J. Roivainen, Performance, Area and Power Trade-Offs in Mesh-Based Emulated Shared Memory CMP Architectures, In the Proceedings of the 2008 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'08), July 14-17, 2008, Las Vegas, USA, 471-477.

[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)

[ITRS09] International Technology Roadmap for Semiconductors, Semiconductor Industry Association, 2009; <http://public.itrs.net/>.

[Jaja92] J. Jaja: Introduction to Parallel Algorithms, Addison-Wesley, Reading, 1992.

[Leppänen96] V. Leppänen, Studies on the realization of PRAM, Dissertation 3, Turku Centre for Computer Science, University of Turku, Turku, 1996.

[Leppänen98] V. Leppänen, Balanced PRAM Simulations via Moving Threads and Hashing, Journal of Universal Computer Science 4, 8 (1998), 675-689.

[Pamunuwa03] D. Pamunuwa, L-R. Zheng and H. Tenhunen, Maximizing Throughput Over Parallel Wire Structures in the Deep Submicrometer Regime, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 11, 2 (April 2003), 224-243.

[Ranade91] A. Ranade, How to Emulate Shared Memory, Journal of Computer and System Sciences 42, (1991), 307-326.

[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.

[Valiant90] L. G. Valiant, A Bridging Model for Parallel Computation, Communications of the ACM 33, 8 (1990), 103-111.

[Vishkin11] U. Vishkin, Using Simple Abstraction to Reinvent Computing for Parallelism, Communications of the ACM 54, 1 (January 2011), 75-85.

Efficient Virtual Machine Scheduling-policy for Virtualized Heterogeneous Multicore Systems

Ibrahim Takouna, Wesam Dawoud, and Christoph Meinel

Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
{ibrahim.takouna, wesam.dawoud, christoph.meinel}@hpi.uni-potsdam.de

Abstract—*Heterogeneous multicore processors could be the future trend of processors' industry due to their performance-power efficiency. In the operating systems domain, A heterogeneity-aware scheduler assigns a thread or an application to an appropriate core to realize this efficiency. Using virtualization technologies enables resource consolidation and achieves effective utilization of resources. Nevertheless, Hypervisors' scheduling-policy is based on the round robin algorithm to ensure fairness among VMs. Emerging heterogeneous system and virtualization increases power savings and enhances resources utilization. This combination needs a new scheduler, which schedules each VM to an appropriate core based on its characteristics. In this paper, we present sources of delay in virtualized environment that could degrade performance of VMs. Then, we investigate the sensitivity of a VM performance to changes in clock frequency. A new scheduling policy was implemented to alleviate sources of delay and to be aware of system's heterogeneity. We emulated our heterogeneous testing environment using DVFS, and we compared the results to default scheduling policy in Hypervisor's scheduler. Nevertheless, the results show performance improvements for VMs that run either CPU-intensive or I/O-intensive applications. Finally, the measured power savings of our heterogeneous testing environment reach up to 25%.*

Keywords: Virtualization, Heterogeneous, Scheduling, Hypervisor, Management.

1. Introduction

Heterogeneous multicore processors could be a common architecture of future multicore processors due to their performance per watt compared to homogeneous processors [1,2,3]. A single processor will contain hundreds of cores that vary in some micro-architecture features such as clock frequency, cache size, area, and others [4], but these cores exploit the same instruction-set architecture. A single chip might have several complex cores and many simple cores. The simple cores are characterized as low-speed clock frequency and low power consumption while fast cores are equipped with high-performance features such as high-speed clock frequency and high power consumption. Consequently, their potential to achieve different levels of performance that meet applications heterogeneity has prompted researchers in

the operating systems domain to implement heterogeneous aware schedulers [5,6,7].

Nevertheless, current Hypervisors' schedulers such as Xen [8] do not support heterogeneous multicore processors, but this issue has been recently tackled in [9]. Authors in [9] have implemented An Asymmetry-Aware Scheduler for Hypervisors (AASH). Using AASH scheduler achieves a good performance improvement for CPU-intensive applications, but this improvement comes with performance degradation for memory and I/O intensive applications. A hypervisor scheduler is considered efficient if it assigns a virtual CPU (vCPU) to run on the appropriate cores based on the application characteristics in terms of CPU-intensive, Memory-intensive, or I/O-intensive. Further, the scheduler must have knowledge of the physical processors' architecture and their characteristics such as cores' clock frequency. By this knowledge, VMs with CPU-intensive applications should be assigned to complex fast cores to be executed faster. Generally, scientific applications are CPU-intensive, multithreaded, and fewer CPU stalls due to infrequent memory accesses or I/O operations. On the other hand, I/O-intensive could be assigned to simple slow cores without losing significant performance and achieving the power savings.

In this paper, we used NAS Parallel Benchmarks [10] as CPU-intensive application and netperf benchmark [11] as I/O-intensive application. We denoted performance sensitivity to CPU clock frequency as "performance-frequency sensitivity" and performance dependency on Domain-0 as "performance-Domain-0 dependency". Our scheduling-policy based on these two categories: "performance-frequency sensitivity" and "performance-Domain-0 dependency" to assign a vCPU to the appropriate core. Consequently, the results showed good performance improvements for VMs with CPU-intensive applications and for VMs with I/O-intensive applications as well. Further, in some experiments, the average combined performance gain reaches up to 70%. Eliminating sources of delay is the foundation of these improvements. Finally, our heterogeneous experimental environment achieves 25% of power savings. The power savings are gained from this architecture, which runs on two cores with high frequency and other two cores with low frequency.

The key contributions are as follows:

- We present and classify sources of delay that might lead

to performance degradation of the whole system such as inter-process commutation and scheduling delay.

- Then, we illustrate performance of NPB benchmark performance-frequency sensitivity for both OpenMP and SERIAL versions of NPB benchmark. Similarly, we investigate sensitivity I/O-intensive applications to clock frequency and their dependency on Domain-0 using netperf benchmark with TCP and UDP streams options.
- We present our modified scheduling-policy that applied to the default credit scheduler of Xen Hypervisor.
- Finally, we discuss the results of experiments showing performance comparison between the default and the modified scheduling-policy.

The rest of paper is organized as follows. The next section discusses sources of delay in virtualized environment and motivates our work. Section 3 presents details about experimental platform and the benchmarks that were used. In Section 4, we discuss sensitivity of VMs performance to CPU clock frequency. Performance evaluation is presented in Section 5. The related work is described in Section 6. Finally, conclusions is presented in Section 7.

2. Background and Motivation

2.1 Background

In virtualized servers, virtual CPUs (vCPUs) of a virtual machine (VM) usually experience scheduling delay due to their competition on physical CPUs (pCPUs) with other vCPUs of the co-hosted VMs including the privileged domain of the Hypervisor (i.e., Domain-0). We discuss sources of delay that impede VMs to achieve the best performance. Sources of delay effect on Network I/O was discussed in [12], meanwhile inter-process communication delay was experienced in [13] due to threads synchronization. We point out these sources as follows.

a) Delay influences network I/O VMs performance:

- 1) The network communication between two VMs is a type of I/O that mainly depends on Domain-0 because a VM does not have privileges to access the physical NIC. Nevertheless, more details could be found in [12].
 - Sending a packet from a VM to another VM in the same host might experience delay due to scheduling Domain-0. The delay is the time period between a VM (a sender) copying a packet into the Domain-0's transmission-I/O-ring and Domain0 being scheduled next to notify another VM (a recipient) in the same host by setting up an event channel notification.
 - Sending/Receiving a packet from a VM to another VM into another host: The delay is the time period between a VM (a sender) copying a packet into the

Domain-0's transmission-I/O-ring and Domain0 being scheduled next to send it via the physical NIC of host and the time period between receiving a packet in the physical NIC of the server hosting the recipient VM and Domain-0 being scheduled next to set up an event channel notification for the recipient VM.

- 2) Delay related to sender VMs scheduling which is the waiting period of a VM to be scheduled for copying a packet into the Domain-0's transmission-I/O-ring.
- 3) Delay related to recipient VMs scheduling which is the duration between when Domain0 sets up an event channel notification for the recipient VM and when the recipient being scheduled next to read the packet from Domain-0's transmission-I/O-ring.

b) Delay influences CPU-intensive VMs performance:

- 1) To achieve better I/O latency, the Xen Credit scheduler prioritizes vCPU I/O-intensive. When a vCPU is blocked waiting for I/O it will not consume credits; when it wakes, it enters the BOOST state and may immediately preempt running vCPU. Generally, vCPU (CPU-intensive) has less credit than vCPU (I/O-intensive). The frequent preemption of vCPU degrades performance especially for cache sensitive applications because some pCPU cycles go for cache-warming.
- 2) Delay related to inter-process communication comes from asynchronous assignment for vCPU. Xen credit scheduler assigns vCPU asynchronously to satisfy the fairness among vCPUs in the host. However, asynchronous scheduling decreases performance of a multithreaded application that needs synchronization among its threads.

After introducing sources of delay, we give an overview of Xen's credit scheduler [8]. The scheduler gives each vCPU 300 credits for a 30ms accounting period. A 100 credits is subtracted from vCPU each tick. A tick equals 10ms. The scheduler selects the next vCPU to run on pCPU after prioritizing vCPUs with either OVER or UNDER according to their remaining credits. The selection is preformed when the current running vCPU finishes its time-slice or its status becomes idle or blocked.

2.2 Motivation

Increasing number of cores in a single chip has become the mainstream industry to avoid vertical scaling of CPU frequency. A single chip expected to contain hundreds of cores that could be heterogeneous either by design or due to variability and possibility of defects with time [14]. Nevertheless, Heterogeneous multicore promises to achieve 60% in power saving compared to homogeneous [3]. Similarly, using virtualization technologies realizes efficient power savings by hosting multiple virtual machines on a single

physical server. In this paper, we combine heterogeneous processors with virtualization technology to demonstrate the potentials of this combination in achieving power savings and maintaining applications performance at the acceptable level. Current Hypervisors' schedulers based on the round robin scheduling algorithm ensure fairness share of physical cores among VMs, but these schedulers were tailored for homogeneous cores, so using them in heterogeneous environment causes large performance losses. However, the advantages of this combination could not be achieved without dynamic classification for VMs and schedulers aware of VMs classification and processors' heterogeneity. This paper sheds light on some of these advantages to motivate research in this area.

3. Testing Environment and Benchmarks

3.1 Experimental Platform

Our experimental platform is a Dell OPTIPLEX 980 server with an Intel quad-core processor frequency range 2.79 - 1.2GHz, 8MB shared L3 Intel Smart Cache, and VID-Voltage rang 0.6500V-1.4000V. The server is equipped with 8GB memory. We emulated a heterogeneous processor according to expected frequency ranges in future heterogeneous systems[4], so we set two cores with high clock frequency $F_F=2.79\text{GHz}$ and other two cores with low clock frequency $F_L=1.33\text{GHz}$. We considered each core as a physical CPU (pCPU). Our experimental virtualized environment based on Ubuntu-10-32bit-Xen 4.1. Ubuntu operating system was used for para-virtualized unprivileged domains. The number of VMs and vCPUs was changed according to experiments purpose, and each experiment in this paper was repeated at least five times and the average of these readings was considered.

3.2 Benchmarks

The NAS Parallel Benchmarks [10] (NPB) was designed to evaluate the performance of HPC systems. NPB consists of a set of programs that differ in dataset size. The dataset size increases according to the chosen class (i.e., S, W, and A-D) during compilation. Further, NPB suite comes in a variety of versions: SERIAL, OpenMP, MPI, and Java. The SERIAL and OpenMP versions were used in our experiments and compiled with class C dataset which is the second largest dataset after class D. Authors of [15] studied NPB characteristics and provided performance analysis for MPI version. Generally, the NPB programs show a high CPU utilization, and this indicates that those programs are computation-intensive and infrequently blocked for communication or I/O operation. When a program runs with four threads, the communication patterns in these programs are as follows. BT and SP exhibit a mesh communication pattern, but BT includes a number of I/O operations. CG

shows a one dimensional nearest neighbor chain pattern. LU and EP show a ring and negligible communication pattern respectively. Finally, CG and LU are communication intensive and their message size is large compared to the other programs, but LU is a synchronous-sensitive among its threads[13]. We denote OpenMP version of NPB by NPB-OMP that encloses CPU-intensive parallel programs, and the SERIAL version by NPB-SER that includes CPU-intensive single thread programs. Furthermore, we refer to individual program in NPB suite using this notion EP-OMP which means EP program of OpenMP version, or EP-SER which means EP program of the SERIAL version.

Netperf [11] is a network benchmark with a variety of options. We used netperf with TCP_STREAM option to measure TCP channel bandwidth and with UDP_STREAM to measure UDP channel bandwidth. In this paper, we refer to them with netperf-TCP and netperf-UDP respectively.

4. VMs SENSITIVITY ANALYSIS

In this section, we analyzed sensitivity of VMs' performance to changes in CPU clock frequency for VMs that run CPU-intensive and I/O-intensive applications. Then, we illustrated dependency of VMs' on Domain-0 for VMs with I/O-intensive applications.

4.1 VMs with NBP Sensitivity Analysis

To analyze VMs performance-frequency sensitivity, we used NPB-SER and NPB-OMP benchmarks as CPU-intensive programs. In this experiment, we pinned vCPUs of Domain-0 to cores (0,1) and vCPUs of VMs were pinned to the another two cores (2,3) to avoid Domain-0's influence on the VMs; in other words, to prevent Domain-0 from being queued with the VMs in the same queue. First, the experiment was run while the cores (2,3) were set to run with high frequency $F_F=2.79\text{GHz}$ as fast cores. Then, it was run again after changing frequency settings of the cores (2,3) to low frequency $F_S=1.33\text{GHz}$ as slow cores. Finally, we used the price elasticity of demand economics formula to determine program's completion time and throughput sensitivity of clock frequency. We considered T the completion time and Th the throughput as the demand, and F clock frequency as the price. The sensitivity was determined using the formula from [18]. $E_{T,F}$ is the completion time sensitivity of clock frequency, and $E_{Th,F}$ is throughput sensitivity of clock frequency.

$$E_{T,F} = \frac{T_F - T_S}{F_F - F_S} * \frac{F_F + F_S}{T_F + T_S} \quad (1)$$

$$E_{Th,F} = \frac{Th_F - Th_S}{F_F - F_S} * \frac{F_F + F_S}{Th_F + Th_S} \quad (2)$$

Due to the inverse relationship between CPU frequency and completion time, $E_{T,F}$ values are negative, so completion time increases as CPU frequency decreases and

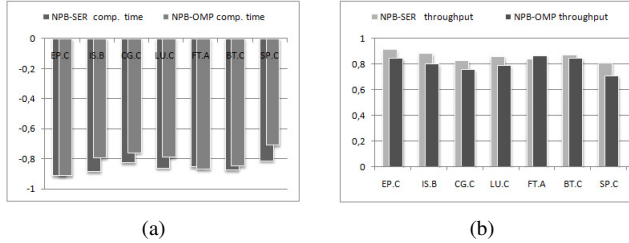


Fig. 1: Performance-frequency sensitivity for NPB-OMP and NPB-SER versions run on a VM with two vCPUs. (a)sensitivity of completion time to frequency, and (b)sensitivity of throughput to frequency.

vice versa. On the other hand, $E_{Th,F}$ values are positive because of the direct relationship between CPU frequency and throughput. Program speedup depends on program characteristics, but it does not have a liner relationship with CPU frequency. However, CPU-intensive programs might have a semi-linear relation with frequency because of either infrequent memory accesses or I/O operations. Figure 1 shows NPB-OMP and NPB-SER benchmarks performance-frequency sensitivity (i.e., completion time and throughput). In NPB benchmark, each program has a different memory access behavior and various inter-process communication patterns. These characteristics determine sensitivity of a program to frequency changes. For example, the completion time of EP-OMP and EP-SER programs had the same and the highest sensitivity. This high sensitivity due to the negligible inter-process communication in the multithreading EP-OMP program, and none inter-process communication in the single thread EP-SER program. Furthermore, EP-OMP is seldom memory access compared with CG-OMP and LU-OMP. Generally, NPB-SER programs sensitivity to frequency changes was higher than NBP-OMP due to the sequential execution of instructions in NPB-SER and inter-process communication patterns or I/O operations in some of NBP-OMP programs such as CG and BT respectively. On the other hand, NBP-OMP programs with intensive inter-process communication were less sensitive to frequency such as CG-OMP and LU-OMP. FT, a mixed type program, almost had the same sensitivity in NPB-SER and NPB-OMP. Unlike LU-OMP, BT-OMP includes a number of I/O operations that do not need synchronization among its threads.

4.2 VMs with I/O Sensitivity Analysis

We analyzed sensitivity of VMs performance with I/O-intensive to CPU frequency. Then, as I/O operations depend on Domain-0, we tested VMs performance-Domain-0 dependency.

4.2.1 CPU Frequency Sensitivity

In this experiment, we ran netperf with TCP-STREAM and UDP-STREAM options to test I/O performance-frequency sensitivity using formula 2. The setting of this experiment was the same setting when we tested VM with NBP sensitivity. As shown in figure 2-(a), TCP test is more sensitive to core frequency than UDP due to the nature of TCP-packet; UDP does neither message fragmentation nor reassembly. Further, the aggregate costs of non-data touching overheads consume majority of the total software processing time. The non-data touching overheads come from as network buffer manipulation, protocol-specific processing, operating system functions, data structure manipulations (other than network buffers), and error checking[16]. To validate our test, we used SCP application TCP-based to transfer a 500MB file between two VMs and we found the same results obtained using netperf-TCP.

4.2.2 VMs with I/O Domain-0 Dependency

In this experiment, we ran netperf benchmark with TCP-STREAM and UDP-STREAM options to test I/O performance-Domain-0 dependency. For this end, we reversed the scenario of VM performance-frequency sensitivity, so the cores (2,3) settings were not changed but were set to high frequency $F_F=2.79\text{GHz}$ where VMs were pinned in cores (2,3). On the other hand, The cores (0,1) were set to high frequency $F_F=2.79\text{GHz}$ where Domain-0 was pinned. Then, we ran it again while the frequency of cores (0,1) is low $F_S=1.33\text{GHz}$. Finally, we computed the performance-Domain-0 dependency using formula (2). The result of this experiment is shown in figure 2-(b). It illustrates that both netperf-TCP and netperf-UDP depend on Domain-0 for commutation between to VMs, but netperf-TCP depends on Domain-0 more than netperf-UDP.

The conclusion is that applications based on TCP protocol are frequency sensitive and they are Domain-0 dependant as depicted in figure 2-(a) and figure 2-(b) respectively.

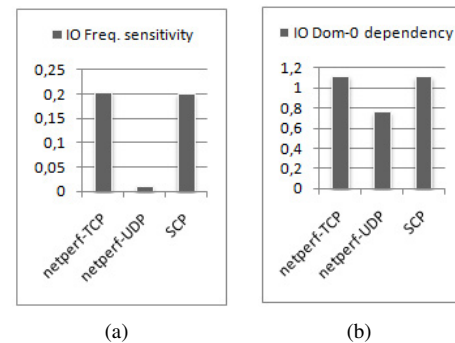


Fig. 2: Performance-frequency sensitivity and Domain-0 dependency for NPB-OMP and NPB-SER versions run on a VM with two vCPUs. (a) performance-frequency sensitivity, and (b) performance-Domain-0 dependency.

5. Performance Evaluations

In this section, we evaluated our improved scheduling-policy with the following rules:

- The weight of VM is proportional to the number of vCPUs.
- CPU-intensive vCPU should not be queued with I/O-intensive vCPU. Furthermore, CPU-intensive vCPU should be placed in the fast pCPU's queue meanwhile I/O-intensive vCPU in the slow pCPU's queue.
- A virtual machine with CPU-intensive application and a single vCPU should be placed in fast pCPU's queue to accelerate the sequential execution.
- The time-slice for the fast pCPU's queue is 30ms and time-slice for slow cores is 10ms as show in figure 3. We chose the value 10ms for the short slice as one tick to avoid high context switching and to keep consistent credit accounting.
- The cores settings for the experiments were that the fast cores (0,1) ran on frequency $F_F=2.79\text{GHz}$ and the slow cores (2,3) ran on frequency $F_S=1.33\text{GHz}$.

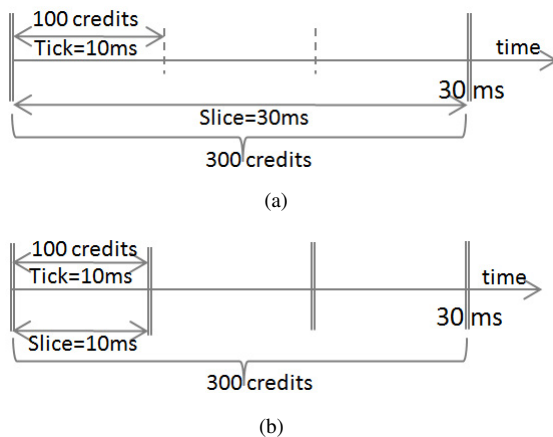


Fig. 3: Scheduling time-slice modifications.(a) time-slice = 30ms for fast cores, and (b) time-slice = 10ms for slow cores. The accounting period of vCPU is 30ms for both fast and slow cores.

5.1 I/O and CPU-intensive Isolation

In this experiment, we created three VMs one with two vCPUs while each of the other two VMs has one vCPUs. We ran netperf on the two VMs with one vCPU for testing TCP and UDP bandwidth channels between them. The VM with two vCPUs used to run NPB-SER and NBP-OMP programs. We ran the three VMs with our new scheduling-policy. First, we used EP and CG programs in NPB-SER with netperf, then EP and CG of NPB-OMP were used. We pinned the VMs with I/O to the slow cores (2,3) and the VM with CPU-intensive was pinned to the fast cores (0,1). Performance improvement for both I/O and CPU-intensive

VMs compared to the default scheduler is illustrated in figure 4. Figure 4-(c) shows that the performance gain of CG.C is better than EP.C. Indeed, EP.C has negligible inter-process communication compared to CG.C which has also memory accesses. On the other hand, netperf-TCP throughput when co-hosted with VM that ran NBP-SER is better than when co-hosted with VM that ran NBP-OMP. As seen in figure 2-(b), netperf depends on Domain-0 and NPB-SER is a single thread test that gave Domain-0 chance to be scheduled in fast cores and improve I/O operations for netperf-TCP. The aggregate average gain is depicted in figure 4-(c). Obviously, isolating CPU-intensive vCPUs from I/O-intensive vCPUs was the main reason for performance improvement. Using isolation eliminated the sources of delay that affect CPU-intensive vCPUs performance.

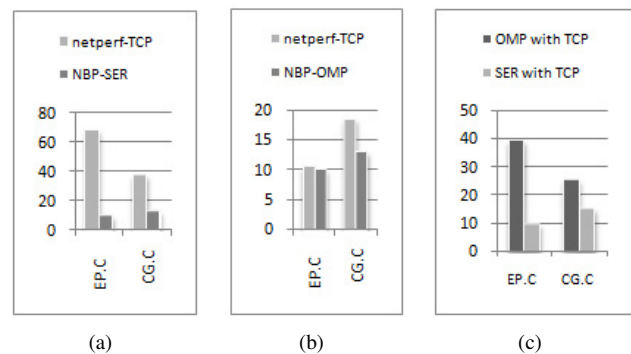


Fig. 4: I/O and CPU-intensive Isolation Performance improvements; netperf-TCP run on a VM with one vCPU, and NPB-OMP run on a VM with two vCPUs. (a) Throughput gain for NPB-OMP and netperf-TCP benchmark, (b) throughput gain for NPB-OMP and netperf-TCP benchmark, and (c) the average improvement of the overall system.

5.2 VMs with sensitive Inter-process Comm.

In this experiment, we tested the performance gain for inter-process communication intensive such as CG and LU of NPB-OMP version. The performance of NPB-OMP benchmark in VM is near to the performance in physical server as long as the vCPUs are less than pCPUs, and LU-OMP is the most sensitive program to communication delay [13]. For testing inter-process communication intensive program performance improvement, we created one VM with one vCPU and another VM with four vCPU. Nevertheless, we had five vCPUs in addition to four vCPUs for Domain-0. The performance gain is illustrated in figure 5 where figure 5-(a) shows Throughput gain and completion time speedup for NPB-OMP while figure 5-(b) illustrates Throughput gain and completion time speedup for NPB-SER. Figure 5-(c) shows the average aggregated performance gain for NPB programs with two versions. Nevertheless, LU-OMP gained about 70% performance improvement. This improvement

due to changing the time-slice of the slow pCPUs' to 10ms which increases scheduling frequency. Increasing scheduling frequency gave chance for inter-process communication and synchronization. Further, decreasing time-slice decreases holding time when vCPU status "busy blocking" holds pCPU [17]. A lot of "busy blocking" wastes pCPU cycles and degrades the overall system performance.

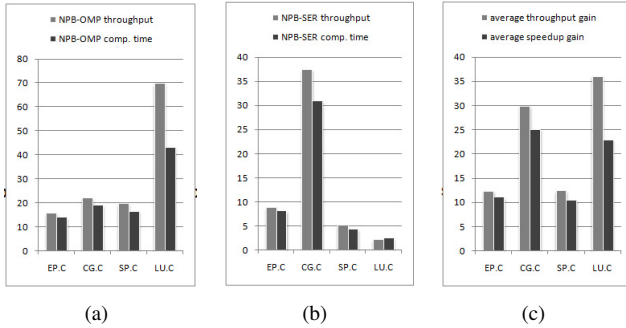


Fig. 5: CPU-intensive with inter-process communication intensive performance improvements: NPB-OMP run on a VM with four vCPUs and NPB-SER run on a VM with one vCPU. (a) Throughput gain and completion time speedup for NPB-OMP, (b) throughput gain and completion time for NPB-SER, and (c) the average improvement of the overall system.

5.3 Estimated Power Savings

As our physical platform is homogeneous, we have used dynamic voltage and frequency scaling (DVFS) mechanism to emulate a Heterogeneous environment. Nevertheless, we calculate the expected power saving in our experimental environment using formula (3). Formula (3) shows that the power consumption has a direct proportional to F which is the clock frequency and the square of Voltage (V).

$$P = C * F * V_{VID}^2 \quad (3)$$

Consequently, homogeneous architecture (HO) will consume P_{HO} with four cores ran on high frequency $F_F=2.79\text{GHz}$ and $V_{VID}=1.4\text{V}$, while heterogeneous architecture (HE) will consume P_{HE} with two cores with high frequency $F_F=2.97\text{GHz}$ with $V_{VID}=1.4\text{V}$ and the other two ran with low frequency $F_S=1.33\text{GHz}$ with $V_{VID}=0.65\text{V}$. The theoretically power savings gain is 45% calculated using $(1 - P_{HE}/P_{HO}) * 100$ formula, but the measured values of power savings reach up to 25%.

6. Related Work

Most of the works related to heterogeneous processors have been done in operating systems domain. Our work overlaps with two categories: (i) Heterogeneous scheduling awareness for OS, (ii) Heterogeneous scheduling awareness for Hypervisors. First, the algorithms for heterogeneous

awareness in operating systems field can be described as follows. The algorithm presented in [6] assigns the best threads (i.e., threads with high computation demands) to run on fast cores; however, selecting the best threads via continuous monitoring of performance based on instructions per cycle (IPC). Furthermore, continuous monitoring for threads before getting the best thread to the fast core might take long time and consume more resources. This algorithm could be modified for Hypervisors if we consider a virtual machine as a long lived thread. By determining the architectural properties of an application, the algorithm in [16] find the best threads to be assigned to fast cores. However, we used the same methodology to classify virtual machines that proposed in [16]. The algorithm proposed in [19] boost the sequential phases of parallel applications by executing them on fast cores. In our work, the scheduler assigns CPU-intensive VMs with single vCPU to fast cores. In [20], the scheduler places more threads on fast cores than slow cores, where the core load is proportional to its frequency speed. Unfortunately, this technique is not suitable for Hypervisors because virtual machines will experience cache contentions that degrade their performance [21].

Second, heterogeneous scheduling awareness for Hypervisors according to the recent work was presented in [9]. These were not much research done in this domain; however, paper [9] was implemented An Asymmetry-Aware Scheduler for Hypervisors which is a scheduler aware to heterogeneity of multicore processor. Using AASH scheduler achieves a good performance improvement for CPU-intensive VMs, but this improvement came with performance degradation for memory and I/O intensive VMs. Our idea of shortening the time-slice for the slow cores similar to dynamic switching frequency scheduling policy proposed in [20], but it was proposed for homogeneous environment and for pinned virtual machines. Authors in [20] suggested to set the time-slice to one millisecond for some CPU-Intensive VMs, but according to the comments in Xen's source code [22], a 1ms is the given delay for pCPU to build its cache for vCPU between vCPU migrations. So, one millisecond time-slice is very expensive for CPU-intensive VMs due to frequent cache-warming that means more pCPU cycles losses.

7. Conclusions

In our scheduling-policy, we invested the recommendations that were proposed for operation systems schedulers. The policy is suitable for virtualized environments that co-hosted heterogeneous type of VMs. We presented scheduling-policy that aware of virtual machines and physical host heterogeneity to realize the promises of Heterogeneous multicore processor in virtualized environments. Analyzing VM characteristics is the most significant stage to place a VM at the suitable cores that keep its performance acceptable. Furthermore, elimination of delay sources that impede performance gaining could bring good performance

improvements. Our results proved that heterogeneous multicore systems could add more advantages in terms of power savings when combined with virtualization technologies. Nevertheless, the average combined performance improvements gain for both CPU-intensive and I/O-intensive VMs reached up to 70% in some experiments compared with default scheduling-policy of Hypervisor's scheduler. Furthermore, the power savings achieved in our experimental environment almost realize the promises in [3], where 25% of power savings have been achieved compared with homogeneous architecture.

References

- [1] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, and P. Husbands, "The Landscape of Parallel Computing Research: A View From Berkeley," UC Berkeley Technical Report UCB/EECS-2006-183, 2006.
- [2] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade, "Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors," *IEEE Computer Architecture Letters* 5(1):4, 2006.
- [3] R. Kumar, K. I. Farkas, and N. Jouppi et al, "Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction," *In Proc. of MICRO 36*, 2003.
- [4] S. Borkar, "Thousand Core Chips-A Technology Perspective," *in Proc. of the DAC*, 2007.
- [5] R. Kumar, Dean M. Tullsen, P. Ranganathan, N. Jouppi, and K. Farkas, "Single-ISA Heterogeneous Multicore Architectures for Multithreaded Workload Performance," *in Proc. of the 31st Annual International Symposium on Computer Architecture*, 2004.
- [6] R. Kumar, D. M. Tullsen, and P. Ranganathan et al, "Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance," *in Proc. of ISCA*, 2004.
- [7] M. Becchi and P. Crowley, "Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures," *in Proc. of the Conference on Computing Frontiers*, 2006.
- [8] 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," *in Proc. SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.
- [9] V. Kazempour, A Kamali, and A. Fedorova, "AASH: an asymmetry-aware scheduler for Hypervisors," *in Proc. of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, 2010.
- [10] R. V. der Wijngaart, "NAS Parallel Benchmarks v. 2.4", NAS Technical Report NAS-02-007, October 2002.
- [11] R Jones, "NetPerf:a Network performance benchmark," <http://www.netperf.org>.
- [12] S. Govindan, A. R. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam, "Xen and co.: communication-aware cpu scheduling for consolidated xen-based hosting platforms," *in Proc. of the 3rd international conference on Virtual execution environments*, pp. 126-136, 2007.
- [13] C. Xu, Y. Bai, and C. Luo, "Performance Evaluation of Parallel Programming in Virtual Machine Environment," *In Proc. of Sixth IFIP International Conference on Network and Parallel Computing*, pp. 140-147, 2009.
- [14] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *in IEEE Micro*, 5(6):10-16, 2005.
- [15] J. Subhlok, S. Venkataramaiah, and A. Singh, "Characterizing NAS benchmark performance on shared heterogeneous networks," *in Proc. of 11th International Heterogeneous Computing Workshop*, 2002.
- [16] J. Kay and J. Pasquale, "The Importance of Non-Data Touching Processing Overheads in TCP/IP," *In Proc. of ACM SIGCOMM*, 1993.
- [17] H. Chen, H. Jin, K. Hu, and J. Huang, "Dynamic Switching-Frequency Scaling: Scheduling pinned Domains in Xen VMM," *in Proc. of 39th International Conference on Parallel Processing*, pp. 287-296, 2010
- [18] D. Shelepov and A. Fedorova, "Scheduling on Heterogeneous Multi-core Processors Using Architectural Signatures," *in Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture*, in conjunction with the 35th International Symposium on Computer Architecture (Beijing, China, June 21-25, 2008). WIOSCA '08.
- [19] J. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A Comprehensive Scheduler for Asymmetric Multicore Processors," *in Proc. of the 5th ACM European Conference on Computer Systems (EuroSys) 2010*, 2010.
- [20] T. Li, D. Baumberger, and D. A. Koufaty et al, "Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures," *In Proc. of SC '07*, pp. 1-11, 2007.
- [21] O. Tickoo, R. Iyer, R. Illikkal, and D. Newell, "Modeling Virtual Machine Performance: Challenges and Approaches," Intel Corporation, 2009.
- [22] <http://lxr.xensource.com/lxr/source>.

Prototyping a Library of Algorithmic Skeletons with Bulk Synchronous Parallel ML

Noman Javed¹, Frédéric Louergue¹, Julien Tesson¹, and Wadoud Bousdira¹

¹LIFO, Université d'Orléans, France

Abstract—*Algorithmic skeletons are a high-level approach to parallel programming that can be combined with widely used programming languages such as Java, C and C++. In this paper we show that prototyping such a library with a structured parallel functional language, namely Bulk Synchronous Parallel ML, provides a parallel implementation with which experiments can be performed and gives some hints about the formal semantics of the library as well.*

Keywords: Parallel programming, Algorithmic skeletons, Functional programming, Applications

1. Introduction

The widespread of parallel machines makes the widespread of structured parallel programming paradigms necessary. Algorithmic skeletons [1], [2], [3] and bulk synchronous parallelism [4], [5] are two such structured parallel models. The former eases the programming: the skeletons (higher-order functions) implement usual patterns of parallel algorithms while corresponding to usual sequential combinators. To reason about the functional semantics of the program is quite similar to reason about the semantics of the program where the sequential combinators are used. The latter eases the design of parallel algorithm by providing a simple and realistic performance model.

Orléans Skeleton Library [6] is an efficient C++ library of parallel algorithmic skeletons that uses expression templates for optimisation. In the design of a new version of OSL, we aim at improving the safety of the library while preserving its expressivity. In this paper we present the prototyping of the new OSL library with a parallel functional programming language that follows the BSP model: Bulk Synchronous Parallel ML (BSML) [7], [8], [9]. Using such a language allows to focus on the design of the algorithms behind the skeletons without having to take into account the C++ mechanisms that can be error-prone, but still being able to run the skeletons in parallel. Bulk Synchronous Parallel ML being currently implemented as a library of the Objective Caml language [10], the skeletons in BSML could be mostly reused in formal development using the Coq proof assistant [11], [12] and form the basis of a formal execution model of OSL.

We first give an overview of programming with BSML (section 2), before explaining the design and prototyping of the library of algorithmic skeletons in BSML (section 3).

Some example applications are implemented using this library and some experiments performed (section 4). We conclude by related work (section 5) and future research direction (section 6).

2. Bulk Synchronous Parallel ML

In the BSP model, the number p of memory-processor pairs of the BSP machine is fixed during execution. p is accessible to the programmer, it is named **bsp_p**. These pairs are interconnected in such a way that point-to-point communications are possible. A global synchronisation unit is available in a BSP computer. The execution of a BSP program is a sequence of super-steps, each one being composed of a phase where each processor computes using only the data it holds, a phase where processors exchange data and a synchronisation barrier that guarantees the completion of data exchange before the start of a new super-step. The other BSP parameters are respectively **bsp_g** (network bandwidth), **bsp_l** (synchronisation time), and **bsp_r** which is a measure of the processors computing power. All parameters, but **bsp_p**, can be obtained by a benchmark program.

BSML is based on a distributed datatype called parallel vector. A parallel vector has type 'a par and embeds p values of any type 'a at each of the p different processors in the parallel machine. The nesting of parallel vectors is not allowed.

The p processors are labelled with naturals from 0 to $p-1$. We use the following notation for a parallel vector:

$$\langle x_0, x_1, \dots, x_{p-1} \rangle : 'a \text{ par}$$

or $\langle x_i \rangle_i$ for short. This vector holds the value x_i at processor i , with all x_i of type 'a. We distinguish this structure from a usual "sequential" vector of size p because the different values, that will be called *local*, are blind from each other. It is only possible to access the local value x_i in two cases:

- 1) locally, on processor i (by the use of a specific primitive), or
- 2) after some communications.

These restrictions are inherent to distributed memory parallelism. This makes parallelism explicit and programs more readable. Since the BSML program deals with a whole parallel machine and individual processors at the same time, a distinction between the levels of execution that take place is needed:

Primitive	Type	Description
$\ll e \gg$	$t \text{ par if } e : t$	$\langle e, \dots, e \rangle$
$\$this\$$ (within a local section)	int	i on processor i
$\$v\$$ (within a local section)	$t \text{ (if } v : t \text{ par)}$	v_i on processor i (if $v = \langle v_0, \dots, v_{p-1} \rangle$)
proj	$'a \text{ par} \rightarrow \text{int} \rightarrow 'a$	$\langle v_0, \dots, v_{p-1} \rangle \mapsto (\mathbf{fun } i \rightarrow v_i)$
put	$(\text{int} \rightarrow 'a) \text{ par} \rightarrow (\text{int} \rightarrow 'a) \text{ par}$	$\langle f_0, \dots, f_{p-1} \rangle \mapsto \langle \mathbf{fun } i \rightarrow f_i 0, \dots, \mathbf{fun } i \rightarrow f_i(p-1) \rangle$

Fig. 1: Summary of BSML Primitives

- *Replicated* execution is the default. Code that doesn't involve BSML primitives (nor, as a consequence, parallel vectors) is run by the parallel machine as it would be by a single processor. Replicated code is executed at the same time by every processor, and leads to the same result everywhere.
- *Local* execution is what happens inside parallel vectors, on each of their components: the processor uses its local data to do computation that may be different from the others.
- *Global* execution concerns the set of all processors together, but as a whole and not as a single processor. Typical example is the use of communication primitives.

BSML programs can be compiled in byte-code, native code or can be evaluated in an interactive fashion using the BSML interactive loop. In this case, when one gives an expression to the top-level, possibly a name, a type and the value of the expression are returned. For example:

```
# bsp_p;;
- : int = 4
```

is the prompt, **bsp_p** is the expression to evaluate, the answer is the second line, giving the name of the value (here no name is given to the value), the type (int) and the value (4). In the remaining of this section, our BSP machine will have 4 processors.

To build a parallel vector containing the same value at all the processors, one can write $\ll e \gg$ where e is a usual "sequential" Objective Caml expression. If the value of e is v then the value of $\ll e \gg$ is the parallel vector $\langle v, \dots, v, \dots, v \rangle$. For example:

```
<< "PDPTA" >> ;;
- : string par = <"PDPTA", "PDPTA", "PDPTA", "PDPTA">
```

There also exists a predefined parallel vector, the value named *this*, that contains the value i at processor i :

```
# this;;
- : int par = <0, 1, 2, 3>
```

The so-called *local section* notation $\ll \gg$ can be used to access the local values of a vector. Let us consider an expression e of type $t \text{ par}$. Being an expression with a parallel type, its value is a parallel vector $\langle v_0, \dots, v_{p-1} \rangle$. Inside a local section, for all processor i , the notation $\$e\$$ represents at processor i the local value v_i . In combination with *this* we have a way to build parallel vectors with different values on the different processors:

```
# let hello = << (string_of_int $this$)^":hello" >> ;;
hello : string par = <"0:hello", "1:hello", "2:hello", "3:hello">
```

Objective Caml being a higher-order functional programming language, it is possible to build parallel vectors of functions:

```
# << ( + ) $this$ >> ;;
- : (int → int) par = <<fun>, <fun>, <fun>, <fun> >
```

Here $(+)$ is the integer addition function in prefix notation, partially applied. At processor i we have the function $\mathbf{fun } x \rightarrow i + x$.

The only way to obtain a sequential value from a parallel expression is the use of the **proj** primitive. The type of this function is $'a \text{ par} \rightarrow \text{int} \rightarrow 'a$. Given a parallel vector, it returns a function such that, applied to the processor identifier of a processor, it returns the value of the vector at this processor. **proj** is often used at the end of a parallel computation to gather the computed results. For example, if we want to convert a parallel vector into a list, we write:

```
# let f = proj hello in List.map f [0; 1; 2; 3];;
- : string list = ["0:hello"; "1:hello"; "2:hello"; "3:hello"]
```

A (almost) total exchange occurs when the **proj** function is applied to its first argument. In BSML this ends the super-step. Note that the communication and synchronisation phases occur only when **proj** is applied to its *first* argument. In the example further applications of f do not imply additional communications and synchronisations.

Some values are considered to be an empty message (or to have size 0) and are thus not communicated through the network, even if the yielded results may suggest they were. It is the case for example for empty lists, the value *None* of the $'a$ option type, *etc.* Thus the communication schema of **proj** may be not a full total exchange if some of the values in the argument vector are values of size 0.

put is the comprehensive communication primitive: It allows any local value to be transferred to any other processor. It is synchronous, and ends the current super-step. Canonical use of **put** is

```
let com = put << fun sendto → e($this$, sendto, $x$)>>
```

where expression e computes (or usually, selects) the data of vector x that should be sent depending on the destination processor sendto . The return value of **put** is another vector of functions. At a processor j the function, when applied to i , yields the value *received from* processor i by processor j .

For example, the following shift function shifts a parallel vector circularly to the right:

```
# let shift v =
  let vdst = << ($this$+1) mod bsp_p >>
  and vsrc = << (bsp_p+($this$-1)) mod bsp_p >> in
  let shifted =
    put << fun dst → if dst = $vdst$ then [$v$] else [] >> in
    << List.hd ($shifted$ $vsrc$) >> ;;
val shift : 'a par → 'a par = <fun>
# shift hello;;
- : string par = <"3:hello", "0:hello", "1:hello", "2:hello">
```

A summary of BSML primitives is given in figure 1.

3. A library of algorithmic skeletons

We implemented a prototype library of a new version of Orléans Skeleton Library in BSML. We first describe the underlying data structure before detailing the main (but not all) skeletons.

3.1 Distributed arrays

The data structure manipulated by the skeletons of our prototype library are distributed arrays. In BSML such a data structure could be implemented as a parallel vector of arrays. However it is convenient, and more efficient, to store some additional information rather than to compute on demand:

- the start index of each processor with respect to the global array: each processor contains one array, but this array is a sub-array of the distributed array considered as a whole array; computing it from the parallel vector of arrays would require communications,
- the global length of the distributed array: if it is not stored then a parallel reduction is needed to compute it from the parallel vector of arrays,
- the distribution: each processor knows the local length of the other local arrays without having to communicate; we represent the distribution as an array of integers of size **bsp_p**.

The type for distributed arrays is thus defined as:

```
type 'a distArray = {
  data : 'a array par;
  startIndex: int par;
  globalSize : int;
  distribution : int array;
}
```

globalSize and distribution are not parallel vector but have usual sequential types. This makes clear that these fields cannot have a different value on two different processors. On the contrary startIndex is a parallel vector: at each processor it contains the start index of the local array in the global array. It is also possible to choose startIndex to be the array of the start indices of all the local arrays. However, for all skeleton but one, it is not necessary for a processor to know the start indices of other processors: therefore it is better to save memory by having only one integer value per processor for startIndex rather than having a *replicated* array of size *p* (than in practice is *p* integer values on each processor).

An example of value, if we have 4 processors, is:

```
# let da = init 11 string_of_int;;
val da : string distArray =
{data = <[["0"; "1"; "2"]], [["3"; "4"; "5"]],
  [["6"; "7"; "8"]], [["9"; "10"]] >;
  startIndex = <0, 3, 6, 9>;
  globalSize = 11;
  distribution = [3; 3; 3; 2]}
```

The set of skeletons provided to the user of the library is given in figure 2. The three first skeletons are, if we take an object oriented programming terminology, constructors of distributed arrays. `make` creates a distributed array of a given size with the given value everywhere, `init` creates a distributed array of a given size with its elements given by applying a function from indices to values, and `atRoot` builds a distributed array from a sequential array at root processor. The two first functions give an evenly distributed array whereas the third one returns a distributed array with values only at processor 0. For the two first constructors, the `startIndex`, `globalSize` and `distribution` fields do not need any communication to be computed as the distribution is known from the global size when the array is evenly distributed (if the size is not divided by the number of processors, the processors with a low process identifier may have one additional element). For the third constructor, communications are required.

3.2 The getPartition and flatten skeletons

The `getPartition` and `flatten` skeletons are such that `flatten(getPartition da) = da`. Basically `getPartition` makes the distribution of the distributed array apparent, and is mostly a change of point of view, that is inexpensive to compute: It just, at each processor, puts the local array into an array of one element. However we wish the `flatten` skeleton to be also inexpensive to compute. Therefore we would like to keep the information related to the global size, start indices and distribution before the `getPartition` to be able to restore them when there is a call to `flatten`. In order to do that, we actually have a fifth field in the `distArray` type: `partitioned`, a boolean. When this field is `true` this means that the `globalSize`, `startIndex` and `distribution` fields refers to a distribution before a call to `getPartition`. The actual distribution could be computed without any communication if we assume that it is a distribution obtained after a call to `getPartition`: the global size is **bsp_p**, the `startIndex` is equal to this and the distribution is such that there is one element per processor.

For example:

```
# let pda = getPartition da;;
val pda : string array distArray =
{ data = <[ [ ["0"; "1"; "2"] ], [ [ ["3"; "4"; "5"] ] ],
  [ [ ["6"; "7"; "8"] ] ], [ [ ["9"; "10"] ] ] >;
  startIndex = <0, 3, 6, 9>; globalSize = 11;
  distribution = [3; 3; 3; 2]; partitioned = true }
```

```

make: int → 'a → 'a distArray
init: int → (int → 'a) → 'a distArray
atRoot: (unit → 'a array) → 'a distArray
getPartition: 'a distArray → 'a array distArray
flatten: 'a array distArray → 'a distArray
map: ('a → 'b) → 'a distArray → 'b distArray
mapIndex: (int → 'a → 'b) → 'a distArray → 'b distArray
zip: ('a → 'b → 'c) → 'a distArray → 'b distArray → 'c distArray

```

```

zipIndex: (int → 'a → 'b → 'c) → 'a distArray → 'b distArray →
           'c distArray
shift: int → (int → 'a) → 'a distArray → 'a distArray
permute: (int → int) → 'a distArray → 'a distArray
balance: 'a distArray → 'a distArray
reduce: ('a → 'a → 'a) → 'a → 'a distArray → 'a
gather: int → 'a distArray → 'a distArray
bcast: int → 'a distArray → 'a distArray

```

Fig. 2: Skeletons of the Prototype Library

However, if other skeletons are applied to a partitioned distributed array, it is not always guaranteed that the distributions can be updated without additional communications. In this case the `startIndex`, `globalSize` and `distribution` fields are replaced by their actual values and the field `partitioned` is set to **false**. A call to `flatten` on a distributed array that is not partitioned incurs communications: each processor contains an array of arrays that is flattened to an array, and the sizes of these arrays are totally exchanged. From these sizes the various fields can be computed (without new communications).

3.3 The map, zip and balance skeletons

The functional semantics of the map skeleton can be written: $\text{map } f [v_0, \dots, v_{n-1}] = [f v_0, \dots, f v_{n-1}]$ where $[v_0, \dots, v_{n-1}]$ is the notation for the data part of a distributed array of global size n . The map skeleton does not change the distribution of a distributed array. However if the distributed array is partitioned (i.e. its field `partitioned` is **true**), the `startIndex`, `globalSize` and `distribution` fields still contain the values they had before the call to `getPartition`. We call this set of fields and their values “the distribution before partitioning”. With a call to `map`, the distribution before partitioning is not guaranteed to be preserved.

For example:

```

let da' =
  let f a = let l = Array.length a in Array.sub a 0 (l/2) in
  map f (getPartition da);;

```

does not preserve the distribution of `da`. If the local sizes of `da` are even, the global size of `flatten da'` will be half of the one of `da`. Moreover the distribution depends on the *local* application of the function `f`, so it is not possible to update the distribution before partitioning without communication. Thus for an application of `map` we remove the distribution information before partitioning (it requires no communication):

```

val da' : string array distArray =
{ data = <[["0"], ["3"], ["6"], ["9"]]>;
  startIndex = <0, 1, 2, 3>; globalSize = 4;
  distribution = [1; 1; 1; 1]; partitioned = false }

```

The `mapIndex` variant of `map` has the same properties. It benefits from having the `startIndex` field. Its informal

functional semantics is:

$$\text{mapIndex } f [v_0, \dots, v_{n-1}] = [f 0 v_0, \dots, f (n-1) v_{n-1}]$$

The zip skeleton is a generalisation of the map skeleton to two distributed arrays. Its informal functional semantics could be written:

$$\text{zip } f [u_0, \dots, u_{n-1}] [v_0, \dots, v_{n-1}] = [f u_0 v_0, \dots, f u_{n-1} v_{n-1}]$$

As the map skeleton, the distribution is preserved, but the distribution before partitioning is not. There is also a problem that may occur with the zip skeleton. In functional programming, this zip function exists and can be applied to two lists that have different sizes. The results will have the length of the smallest input list. However in a distributed settings, this is not so easy as even same global sizes may correspond to unaligned distributed arrays. Therefore to have a safe library, we check that the two distributed arrays have the *same distribution*: if not, an exception, corresponding to a programming error, is raised. This is not a limitation as we provide a balance skeleton. This check does not imply any communication as the distributions are stored in the field `distribution`.

The balance skeleton changes the distribution of a distributed array to an even distribution: communications are required if the distributed array is not already evenly distributed. It is to be noticed that partitioned arrays are actually evenly distributed (there is one element per processor). Therefore the balance skeleton preserves the distribution before partitioning.

3.4 The shift skeleton

The shift skeleton is used for communications. It requires an offset d , a replacement function f , and a distributed array $[v_0, \dots, v_{n-1}]$. Its informal functional semantics follows

$$\begin{aligned} & \text{shift } d f [v_0, \dots, v_{n-1}] \\ &= [f(0); \dots; f(d-1); v_0; \dots; v_{n-d-1}] \quad \text{if } d > 0 \\ & \text{shift } d f [v_0, \dots, v_{n-1}] \\ &= [v_d; \dots; v_{n-1}; f(n-d-1); \dots; f(n-1)] \quad \text{if } d < 0 \end{aligned}$$

The shift skeleton preserves distribution, but it does not preserve distribution before partitioning: the sizes of the values generated by the replacement function are known only locally to the processor where they are produced.

No communications are needed if d is 0 or $|d|$ is greater than the global size of the distributed array. In the first case shift is identity, in the second case the application shift d f is equivalent to $\text{mapIndex } f'$ where $f' i x = f i$.

If the shift skeleton is easy to write when the distribution is even, it is a bit more complicated when the distribution can be any distribution, and there are “holes” in the distribution. We will illustrate the different steps on the following distributed array for a shift with 3 as offset:

```
val da : string distArray =
{ data = < ["A"; "B"], ["C"; "D"], [], ["E"; "F"; "G"] >;
  startIndex = <0, 2, -1, 4>; globalSize = 7;
  distribution = [[2; 2; 0; 3]]; partitioned = false }
```

It proceeds has follows.

At each processor, we compute the sub-array of elements to be communicated (shifted) to other processors, information about distribution is necessary to do so. As at each processor we have the starting index of the local array in the parallel vector `startIndex`, we can compute locally the global destination index of each element to be shifted to other processors: it is the current global index plus the offset. Then for each element of this sub-array, we should compute the destination processor that corresponds to this global destination index: (a) from the distribution array, we compute the prefix sum of the distribution (the start indices) where the processors with no elements have been filtered out, each value being paired with its corresponding processor. From example, if the distribution is the one of `da`, we obtain: $[(0, 0); (1, 2); (3, 4)]$; (b) then using a binary search on this array it is possible to obtain the destination processor (first component of the pair) for each global destination index in time $\mathcal{O}(\log_2 p)$; (c) the values to be sent to the same processor are packed together.

With the distributed array `da` above, we obtain the following parallel vector:

```
- : (int * string array) list Bsm1.par =
< [(1, ["A"]); (3, ["B"])], [(3, ["C"; "D"])], [], [] >
```

We use a variant of the BSML `put` primitive, that takes as input a parallel vector of lists of pairs (destination, message) instead of a parallel vector of functions, for the communications. At the end of this step we obtain the following parallel vector:

```
string array Bsm1.par = < [], ["A"], [], ["B"; "C"; "D"] >
```

Finally we perform a local shift whose replacement function either calls the global shift replacement function or returns elements communicated in the previous step. In the example, the global replacement function is called on processor 0 and 1 (for the first element), and the elements are taken from the previous parallel vector of arrays for processor 1 (second element), and processor 3 (for the 3 last elements):

```
# shift 3 (fun i → "Nothing") da;;
```

```
- : string distArray =
{ data = < ["Nothing"; "Nothing"], ["Nothing"; "A"],
  [], ["B"; "C"; "D"] >; ... }
```

3.5 The permute skeleton

The permute skeleton takes as input a bijective function from 0 to the global size of its distributed array argument. Its informal functional semantics could be written as:

$$\text{permute } f [v_0, \dots, v_{n-1}] = [v_{f^{-1}(0)}, \dots, v_{f^{-1}(n-1)}]$$

The permute skeleton is based on the same auxiliary functions than the shift skeleton. For each element of the distributed array, we compute: first its global destination index, obtained by applying the bijection f to the global index, then we compute the destination processor according to this global destination index, in the same way we do for the shift skeleton.

One concern with the permute skeleton is to check whether the function f is bijective or not. One possibility is to perform the check independently on each processor, applying the function to all possible indices: this would requires $\mathcal{O}(n)$ operations at each processor, compared to the $\mathcal{O}(\frac{n}{p})$ applications (if the distributed array is evenly distributed) of f needed to compute the destination processor. This may be quite costly if $p \ll n$. Moreover, we have written the code in such a way that if f is not bijective, no run-time error will occur but the global size of the obtained distributed array will not be the same than the original size. Therefore we can perform a total exchange of the local sizes, then compute the global size and compare it to the initial global size to check if the function was actually a bijection (and raise an exception to indicate the programming error if there is one). The additional cost of this check is $\mathcal{O}((p-1) \times g + L)$. Thus we could dynamically determine, depending on the BSP parameters of the machine, if the first version of the check is more expensive or not than the second version of the check, and choose accordingly the best version. In the current prototype only the second version is performed.

Unlike shift, it is possible with the permute skeleton to update both the actual distribution and the distribution before partitioning without additional communications.

4. Examples and experiments

4.1 Heat Equation 1D

The diffusion of heat in a bar of metal is governed by the following discretised equation:

$$\begin{cases} h(x, t+dt) = \gamma(h(x+dx, t) + h(x-dx, t) - 2h(x, t)) \\ \quad \quad \quad + h(x, t) \\ h(0, t) = l \\ h(1, t) = r \end{cases}$$

where $h(x, t)$ is the temperature in the bar at position x at time t , l and r are the temperatures outside the bar (boundary

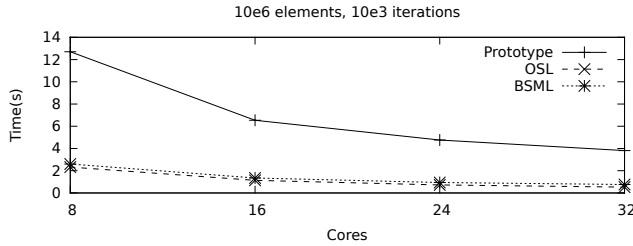


Fig. 3: Heat Diffusion Simulation

conditions), $\gamma = \frac{\kappa dt}{dx^2}$ where κ is the heat diffusivity in the metal, and dx and dt are respectively the space and time steps of the discretisation.

If the temperature in the bar is represented by a distributed array of floating point numbers, the update of the temperature is performed by the following function:

```
(* step: float → float → float → float → float →
   float distArray → float distArray *)
let step kappa dt dx l r bar =
  let barR = shift 1 (fun _ → l) bar
  and barL = shift (-1) (fun _ → r) bar in
  zip
    (fun vl ul → (kappa *. dt) /. (dx *. dx) *. (vl -. 2. *. ul) +. ul)
    (zip (+.) barR barL)
  bar
```

Simulating the diffusion of heat in the bar of metal is then an iterative application of the function `step`.

We have run three different versions of a one dimensional heat diffusion simulation:

- the above version written using the proposed prototype library,
- a version written directly using BSML primitives, without using the proposed library: the programming style is a bit lower level than the previous one and we choose the most efficient version of the programs presented in [8],
- a version written in C++ using the OSL library [6], whose code uses mostly the same skeletons than the first version.

The experiments were conducted on the SPEED parallel machine of the university of Orléans: it is a shared-memory machine with 4 AMD Opteron 6174 processors, each being a 12 cores processor. We set 10^6 discretisation points for the bar of metal, and run 10^3 time steps. The timings are presented in figure 3.

The OSL and BSML versions perform almost the same, the BSML version being less efficient with respect to the OSL version as the number of processors increases. This is due to the fact that the communication overhead is higher with BSML. The version implemented using the prototype BSML implementation of OSL is less efficient: this is due to the fact that quite many intermediate arrays are created. However the difference is less than one order of magnitude:

it is still reasonable to experiment parallel runs with this prototype. In a real implementation of the new OSL library these intermediate copies will be removed using the C++ expression templates optimisation technique.

4.2 Sorting

To show the expressiveness of our library, we implement a (bulk synchronous) parallel regular sampling sort [13]. We assume that there are at least $\mathit{bsp_p}-1$ elements on each processor, that the array is evenly distributed among the processors, and that the elements are distinct. It is not a limitation since elements could be made distinct by transforming each element into a pair composed of the index of the element in the distributed array and the initial value. Moreover the distributed array can be evenly distributed using the balance skeleton.

The comparison functions we use in this program are such that applied to two values v_1 and v_2 , the result is negative if v_1 is smaller than v_2 , is zero if the values are equal, and positive otherwise.

If we assume to have the following sequential functions, the parallel regular sampling sort can be implemented as shown in figure 4:

- `sortArray: ('a → 'a → int) → 'a array → 'a array` takes a comparison function and returns a sorted version of the array argument,
- `getSamples: 'a array → 'a array` returns $\mathit{bsp_p}-1$ samples, regularly taken from its array argument,
- `mergeArrays: ('a → 'a → int) → 'a array array → 'a array` takes a comparison function, and an array of sorted arrays, and it returns the sorted array obtained by merging the input arrays,
- `cut: ('a → 'a → int) → 'a array → 'a array → 'a array array` cuts the first array argument into pieces according to the samples of the second array argument: there is one more piece than the number of samples
- `compose` is usual function composition.

5. Related work

Muesli [14] and SkeTo [15] are two libraries of algorithmic skeletons for C++. Both have skeletons on distributed arrays but the sets of skeletons differ from the one presented here: `permute` is not available in SkeTo, `shift` is not available in Muesli. Our `getPartition`, `flatten` and `balance` skeletons, combined with more classical skeletons offer an expressivity that can not be attained with Muesli or SkeTo that are limited to evenly distributed arrays. However both libraries also offer skeletons for other data structures than distributed arrays such as dense or sparse matrices.

[16], [17] also use BSML to implement skeletons, but with a different set and a different encoding of distributed arrays that do not contain a local and uniform representation of the distribution. As Muesli and SkeTo, the set of skeletons provided restrict the distribution of the distributed arrays that


```
(* sort: ('a → 'a → int) → 'a distArray → 'a distArray *)
let sort cmp da =
  let partitions = map (sortArray cmp) (getPartition da) in
  let fstSamples = map getSamples partitions in
  let sndSamples = bcast 0 (map (compose getSamples (mergeArrays cmp))(getPartition (gather 0 fstSamples))) in
  let pieces = flatten (zip (cut cmp) partitions sndSamples) in
  flatten (map (mergeArrays cmp) (getPartition (permute (fun i → (i/bsp_p)+bsp_p*(i mod bsp_p)) pieces)))
```

Fig. 4: Regular Sampling Sort

can be built. By contrast we allow the construction of a larger variety of distributed arrays while preserving the safety of the library. The parallel functional programming language Eden [18] is also often used to implement algorithmic skeletons [19], [20]. Its model of parallelism is quite different from the parallelism offered by BSML.

6. Conclusion and future work

In this paper we have presented a prototype implementation of a skeleton library written with the structured parallel functional programming language BSML. As it is possible to prove the correctness of BSML programs within the Coq proof assistant, we could prove the correctness of such a library, the code of the skeletons being almost the same in Coq and BSML. Although implementing a library with such a functional language makes possible the reuse for formal semantics development, there are also parallel implementations of BSML that allow the parallel execution of programs written with this skeleton library. The experiments we performed show the limits in term of efficiency, compared to a direct implementation of the examples in BSML, or in a C++ skeleton library.

Future work include the design of a formal execution model and the implementation of new version of the Orléans Skeleton Library for C++, making it safer and more efficient. This model and implementation will both benefit from the lessons learnt while building the prototype.

Acknowledgements

This work is supported by the *Agence Nationale de la Recherche* through the project “Parallel Programming Development with Algorithmic Skeletons” (PaPDAS). The SPEED machine was funded by the *Conseil Général du Loiret*. Noman Javed and Julien Tesson are respectively supported by grants from the Higher Education Commission of Pakistan and from the French Ministry of Research.

References

- [1] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989, available at <http://homepages.inf.ed.ac.uk/mic/Pubs>.
- [2] S. Pelagatti, *Structured Development of Parallel Programs*. Taylor & Francis, 1998.
- [3] F. A. Rabhi and S. Gorlatch, Eds., *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.
- [4] L. G. Valiant, “A bridging model for parallel computation,” *Comm. of the ACM*, vol. 33, no. 8, p. 103, 1990.
- [5] W. F. McColl, “Scalability, portability and predictability: The BSP approach to parallel programming,” *Future Generation Computer Systems*, vol. 12, pp. 265–272, 1996.
- [6] N. Javed and F. Loulergue, “OSL: Optimized Bulk Synchronous Parallel Skeletons on Distributed Arrays,” in *8th international Conference on Advanced Parallel Processing Technologies (APPT'09)*, ser. LNCS 5737, Y. Don, R. Gruber, and J. Joller, Eds. Springer, 2009, pp. 436–451.
- [7] F. Loulergue, F. Gava, and D. Billiet, “Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction,” in *International Conference on Computational Science (ICCS)*, ser. LNCS 3515, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Springer, 2005, pp. 1046–1054.
- [8] W. Boudsira, F. Gava, L. Gesbert, F. Loulergue, and G. Petiot, “Functional Parallel Programming with Revised Bulk Synchronous Parallel ML,” in *First International Conference on Networking and Computing (ICNC 2010), 2nd International Workshop on Parallel and Distributed Algorithms and Applications (PDAA)*, K. Nakano, Ed. IEEE Computer Society, 2010, pp. 191–196.
- [9] The BSML Development Team, “The BSML Library version 0.5,” <http://traclifo.univ-orleans.fr/BSML>, august 2010.
- [10] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, “The Objective Caml System release 3.12,” <http://caml.inria.fr>, 2010.
- [11] The Coq Development Team, “The Coq Proof Assistant,” <http://coq.inria.fr>.
- [12] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development*. Springer, 2004.
- [13] H. Shi and J. Schaeffer, “Parallel sorting by regular sampling,” *Journal of Parallel and Distributed Computing*, vol. 14, pp. 361–372, 1992.
- [14] P. Ciechanowicz and H. Kuchen, “Enhancing Muesli’s Data Parallel Skeletons for Multi-core Computer Architectures,” in *IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2010, pp. 108–113.
- [15] K. Matsuzaki and K. Emoto, “Lessons from Implementing the BiCGStab Method with SkeTo Library,” in *4th workshop on High-Level Parallel Programming and Applications (HLPP)*. ACM, 2010.
- [16] F. Gava and I. Garnier, “New implementation of a BSP composition primitive with application to the implementation of algorithmic skeletons,” in *23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009), APDCM workshop*. IEEE, 2009, pp. 1–8.
- [17] F. Gava and S. Tan, “Implémentation et prédiction des performances de squelettes data-parallèles en utilisant un langage BSP de haut niveau,” in *Journées Francophones des Langages Applicatifs (JFLA)*, ser. Studia Informatica Universalis, S. Conchon and A. Mahboubi, Eds. Hermann, 2011, pp. 39–65.
- [18] R. Loogen, Y. Ortega-Mallen, and R. Pena-Mari, “Parallel functional programming in eden,” *Journal of Functional Programming*, vol. 3, no. 15, pp. 431–475, 2005.
- [19] J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen, “Parallel FFT with Eden Skeletons,” in *Parallel Computing Technologies*, ser. LNCS 5698, V. Malyshev, Ed. Springer, 2009, pp. 73–83.
- [20] M. Dieterle, J. Berthold, and R. Loogen, “A Skeleton for Distributed Work Pools in Eden,” in *10th International Symposium on Functional and Logic Programming*, ser. LNCS 6009, M. Blume, N. Kobayashi, and G. Vidal, Eds. Springer, 2010, pp. 337–353.

A Parallel Architecture Using HDF for Storing DICOM Medical Images on Distributed File Systems

Tiago Steinmetz Soares
Informatics and Statistic Department
Federal University of Santa Catarina
Florianopolis, Brazil
 Email: steinmetz@telemedicina.inf.ufsc.br

Douglas D.J. de Macedo
Post-Graduate Program of Knowledge Engineering and Management
Federal University of Santa Catarina
Florianopolis, Brazil
 macedo@inf.ufsc.br

Michael A. Bauer
Department of Computer Science
University of Western Ontario, UWO
London, Canada
 bauer@uwo.ca

M.A.R Dantas
Informatics and Statistic Department
Federal University of Santa Catarina
Florianopolis, Brazil
 mario@inf.ufsc.br

Abstract—The Hierarchical Data Format (HDF) is an interesting approach for developing scientific applications where a large amount of data must be stored and accessed. A telemedicine project underway in the State of Santa Catarina (SC), in Brazil, has developed a server called the CyclopsDCM-Server, which adopts the HDF for the manipulation of medical images (DICOM). This paper proposes a new approach for the parallel implementation of I/O operations for the medical images stored on this server. This effort was based upon the MPI paradigm that is supported by the version 5 of the HDF. Early experiments indicate that the proposed approach can achieve very good performance when compared to the standard HDF implemented in the CyclopsDCM-Server.

Keywords—Parallel I/O; HDF5; DICOM; Telemedicine; PVFS; MPI;

I. INTRODUCTION

The success of an interactive telemedicine prototype experiment varied in the 1960s [1], between the Massachusetts General Hospital and a medical station at Bostons Logan International Airport, led to a dissemination of the idea of telemedicine throughout several countries. The term telemedicine is commonly used to refer to the remote delivery of health care, basically providing specialized health care, medical diagnosis and monitoring through telecommunications technology to people who cannot access to a medical system directly [2]. However, the capabilities of the equipment must be transmitted at least equal in quality to the information transmitted in the traditional setting. Indeed, the capabilities of the technology are expanding rapidly, becoming faster, more efficient and cheaper, enabling lower costs for the implementation and growth of telemedicine systems.

Researchers from the Telemedicine Laboratory at UFSC [3], adopting the telemedicine approach, created a telemedicine network project called Rede Catarinense

de Telemedicina (RCTM). This project aims to provide connections among different hospitals and different cities within the State of Santa Catarina to provide access to exams, electrocardiograms (EKG), and images from magnetic resonance, computed tomography, X-ray angiography and nuclear medicine [4]. All information acquired of a patient is sent online as DICOM images (Digital Image Communications in Medicine) to a developed PACS (Picture Archiving and Communication Systems), the CyclopsDCM-Server [5] and could be retrieved anytime where the system is deployed.

CyclopsDCMServer is a DICOM medical image facility that was conceived by the Cyclops Group [6] to provide DICOM image storage and wide area network (WAN) access. The CyclopsDCMServer stores all information in a ordinary data base PostgreSQL and can handle around 8 terabytes [4]. Summarizing, this server provides segmentation service for the incoming information, processing and storing the images in a centralized database.

A new DCMServer architecture was proposed to circumvent some of the issues of ordinary relational databases and it has been improve since then. This architecture has two basic applications, PVFS and HDF5. PVFS (Parallel virtual file system) is a distributed file systems designed to scale to petabytes of storage and provide high access rates [7]. The Hierarchical Data Format 5 (HDF5) is a data model for high volume and complex data.

This paper is organized as follows. We start by describing DICOM images (Section 2) and some HDF5 definitions (Section 3), followed by some background about from previous work done on the system in Section 4; this describes several important aspects of the project. In the Section 5 we present some related work and in Section 6 the proposed architecture. In Section 7 we present some experimental

results, with subsections related to the environment and experiments. Finally in the Section 8 we present the conclusions and future work.

II. DICOM IMAGES

The Digital Imaging and Communications in Medicine standard is one of the most universal and fundamental standards in digital medical imaging. The DICOM standard was defined in 1992, and was the third version of the ACR-NEMA Standards Publication PS3. Before this standard was established, each manufacture created their own solution for visualization, storage and impression of digital images. The first ACR-NEMA standard was conceived in 1983 by the American College of Radiology, with main principle to make digital medical images independent of device manufactures, creating a unique standard for medical devices and facilitating the expansion of digital images [8].

Taking many important features from earlier and other standards, the early versions of focused on the improvement and correction of some issues, and where those publications provided specifications related to hardware interfaces, it introduced a set of data format and commands for software packages. Completed in September 1992, the third version came with major revision, supplying increasing variety of digital devices and their communications protocols. This version was called DICOM 3.0, as it followed two earlier ACR-NEMA editions; the standard is reviewed annually and updated with new supplements if necessary [8].

Another important subject relative to DICOM is the Picture Archiving and Communication Systems. PACS consists in hardware and software medical systems designed to run digital medical imaging and is supported by major medical imaging equipment manufacturers. It embraces digital image acquisition devices, digital image archives and workstations.

The CyclopsDCMServer is both a digital image archive system and workstation, which was developed by the Cyclops Group. Created to work with PACS equipment as hospitals and radiology clinics, the purpose of the server is to store and retrieve DICOM index files from an ordinary data base managed by a relational DBMS, such as PostgreSQL. All communication between the server and medical equipment is performed through TCP/IP.

Nowdays, the server supports eight of the several DICOM modalities, namely: computed radiography (CR); computed tomography (CT); magnetic resonance (MR); nuclear medicine (NM); ultrasound (US); X-ray angiography (XA); electrocardiograms (DICOM waveform); DICOM structured reporting (SR) [5].

III. HIERARCHICAL DATA FORMAT (HDF)

Developed by the HDF group at the University of Illinois, initially in the 90s, the goal of HDF is to support large scientific data; the current version is HDF5. One of the main feature of HDF5 is that files can contain binary data as

multi-dimensional arrays and allow direct access to parts of the file without first parsing the entire contents [9]. HDF5 is designed for storing large scientific data, including high performance data manipulation supporting random access, number encoding in native format, data compression, individual data set encryption, and storage strategies for parallel I/O and multidimensional data structures.

There are two essential structures in HDF5 which forms the base for the library: dataset and group. Dataset is a multi-dimensional array of datatype; HDF stores and organize all kinds of data from atomic to composed types, similar to the C struct construct. Other special array operations, such as chunks, compression and extendability, are available through the HDF library and can be applied to a dataset. The group is similar to UNIX directories, though cycles are allowed. Every file is started with a root group, represented as /, and could be followed by the name of another group or a dataset.

An important feature of HDF5 is support for standard parallel I/O interfaces. The Parallel Hierarchical Data Format 5 (Parallel HDF5) required MPI/IO interface through MPICH ROMIO [10] or a vendors MPI-IO, but it does not offer compatibility with shared memory programming. Implemented to get better performance in I/O procedures, the Parallel HDF5 uses distributed file system, such as Parallel Virtual File System (PVFS), Lustre, GPFS and specially configured NFS.

The idea of Parallel HDF5 is to make it easy for users to use the library and provide compatibility with serial HDF5 file. One approach is to read and write data by hyperslab [9], i.e., a multidimensional array that can be spread by rows, columns, patterns and chunks, and a hyperslab selection could be a logically contiguous collection of points, or it can be a regular pattern of points or blocks, depending on the type used. Other important structure is the dataspace. Through a dataspace required components of dataset or even a attributes are defined, as well as array ranks, sizes and types. The difference between the types of hyperslabs is the way that each process will access data of the datasets.

IV. BACKGROUND

One project that has been underway at the Telemedicina Laboratory at UFSC since 2008 is to explore new architectures for DICOM images using distributed file systems. The purpose of this research is to address issues of telemedicine environments based on ordinary database systems. These include addressing issues such as scalability, information distribution, ability to use high performance system techniques and operational costs. Among some of the procedures used to avoid the scalability issue, the project design was to use high performance distributed systems, like clusters or grids [4]. The first approach taken apart from the usual system was to store all information hierarchically, namely, organize and store in HDF5 data format. The second step was to use PVFS as a distributed file system.

Since the DICOM server normally supports drivers only for standard DBMSs, it was necessary to create something similar to these drivers. The HDF5 Wrapper Library (H5WL) was created for this purpose. As the name suggests, this library contains a wrapper object which is used to create, locate, collect and store information related to DICOM images using HDF5 files. When the CyclopsDCMServer requires a creation of new HDF5 file for H5WL, this is created using PVFS.

Two important entities were introduced to help reading data. The first entity consists of information related to the image, such as name of the patient, dimension of the image and other characteristics. The second is the image entity which represents the binary information created by PACS equipment (e.g., computed tomography (CT) or magnetic resonance (MR) images).

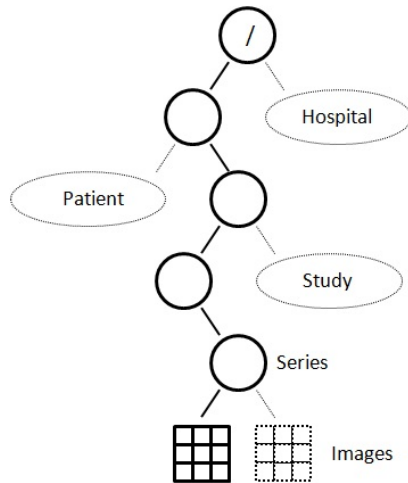


Figure 1. Hierarchical data structured [4]

Another feature is how the hierarchical data structured is organized. An example of the structure is shown in Figure 1, where the data is organized around six layers: root, hospital, patient, study, series and image.

When a client sends an image to the CyclopsDCMServer, the image is captured and the hierarchy is created through calls to the H5WL methods, creating groups that provide information that is used to identify the image inside the file. The biggest component of a DICOM file is the image, which is the leaf of the structure. Basically, the structure image is a compressed image (JPEG) [8] and is responsible for representing the DICOM image.

This architecture, proposed by Macedo [1], has virtues and weakness. Based on 25 experiences with the system, it showed an average improvement in storage of about 16% when compared with the usual system using DBMSs.

However, in term of retrieval operations, there was a drop in performance, with average decrease in performance of around 21%. This was due to mechanisms used to provide

similar behavior to that of standard DBMSs when retrieving information. This paper addresses this problem by proposing an extension to the architecture to take more advantage of a parallel environment. This architecture is detailed in Section VI.

V. RELATED WORK

There is little published work in the telemedicine field which uses HDF5 to store images, as most medical image servers come with drivers only for ordinary data bases. The three works below are similar to the current work in that they use parallel I/O as a solution for I/O bottlenecks access for large amounts of stored data.

The research work presented in Nikhil Laghave [11], is very similar to our work. This work is focused on the use of a parallel I/O library for scalability issues involving fermion dynamics for nuclear structure (MFDn). This work used the HDF5 parallel version for parallel I/O, testing with collective and independent models. As result, there was a gain in the efficiency of input/output of large datasets and the cost of using parallel I/O was less than sequential I/O for sufficiently large datasets.

In particle-based accelerator simulation groups, it is possible to find some HDF5 work. The work of A. Adelman [12] focused on using parallel I/O for particle simulations which involved vast quantities of data and dimensional arrays. He used parallel I/O performance for MPI code as well parallel HDF5. He compared read and write performance in simulations between Parallel HDF5, mpi-io and one file per process. HDF5 showed good performance in writing, though mpi-io showed better results.

H. Yu [13] presented interesting work, though he did not use parallel HDF as solution for his problem, but rather a similar paradigm. His works dealt with large earthquake simulations which require terabytes of storage space and encountered I/O bottleneck issues. He developed his own parallel I/O strategies through MPI I/O to address his needs and was able to remove the I/O bottleneck and also hide pre-processing costs.

VI. PARALLEL ARCHITECTURE

One of the great features available in HDF5 and was not considered in Macedos approach, is the support for MPI communication for parallel processing. As a possible solution to the bottleneck of retrieving data, we provide additional features using the Parallel HDF5 library. The main propose is to get better performance using parallel data access to HDF files stored in the PVFS distributed file system. The Parallel HDF5 library requires a parallel MPI/IO interface and, when working with MPI, it is necessary to design it to be used in a cluster environment. Another important requisite is the necessity to use the mpirun shell script to run any MPI application, which attempts to hide the differences in starting jobs for various devices from the user

[14]. For this, it is necessary to create a additional procedure to work with the CyclopsDCMServer. This procedure should be called every time when is required to retrieve or store some medical information.

It is noteworthy that Parallel HDF5 has support for PVFS through MPICH ROMIO. In this case, our first task is to build the environment using MPICH2 [12] with ROMIO for PVFS. With the environment built, the second task is to create the application which it will be responsible for reading and writing a dataset into a file.

Figure 2 illustrates how the architecture works. The functionality, basically, is the same as described in the previous section, the difference is inside of the H5WL. Instead of having the H5WL responsible for reading and writing the binary information created by PACS, it will be treated as a new parallel application. The parallel application will be initiated by H5WL calling mpirun shell script.

Independent of a read or write function, when H5WL calls the MPI application, all communication between them will be made by socket connections. The communication is done by the master process (represented by MPI process zero) and H5WL for passing function parameters like the location that is the target of an operation (group path), the image buffer and the number of MPI processes. The difference between the functions is the way that the server and MPI applications will communicate. For write functions, the H5WL will first receive the DICOM file, create a new hierarchy of the image based on DICOM file layers (Figure 1), get the path location for new image (JPEG image) and then call mpirun procedure to start the MPI application. The MPI application has functionally to read and write a buffer, without being concerned whether the image exists or not, as that is the responsibility of H5WL. The master process will first communicate with H5WL to retrieve the function to perform, get the location (group) of image in the HDF5 structure and the arguments for the job. If it is a write function, it will need the stream of images to be stored. In case of a read function, the application will only need the path group as parameters, and it will return to server all buffers read from the HDF file.

Independent of the job, the master process has to define the access properties, model and size for each process. The Parallel HDF5 library has available two types of properties (collective and independent data access) and four hyperslab model (Contiguous Hyperslab, Regularly Spaced Data, Pattern and Chunk) [9]. Then the master node has to distribute the memory buffer (write function) or file location to each process. Finally, once the jobs have executed, the main process will return to the wrapper the status of reading or writing the buffer.

VII. EXPERIMENTAL RESULTS

Our experiments are based on the Parallel HDF5 architecture, adapted to use PVFS, and sequential CyclopsD-

CMSServe. The Parallel HDF5 properties used for read and write on the MPI application is: independent data access model and Contiguous Hyperlasb, which entail distributing the buffer by rows as show in the Figure 3.

It is important to note that our results do not take into consideration external factors, like computers using the same network

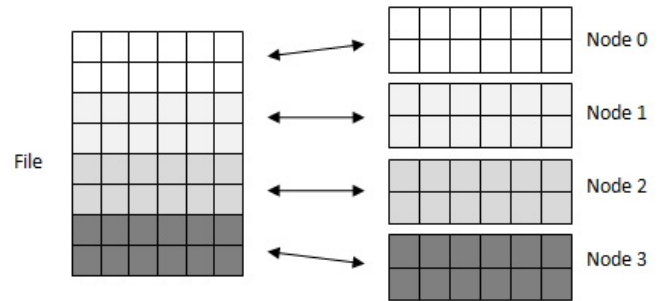


Figure 3. Contiguous Hyperlasb

A. Environment

The environment used for the experiments consist of a four node cluster, as specified in Table 1. The cluster is non-dedicated and is used just for experiments and belongs to Telemedicine Laboratory. The connection network is 100 Mbs Ethernet. The operating system installed on all nodes is CentOS with kernel 2.6.18. PVFS is used on only one metadata node. Each has a PVFS client for access to the PVFS file system and each node is also a MPI executioner and has an MPI application in its own file system for access to HDF file found in Parallel Virtual File System.

Name	CPU	Memory	HD
Node1	AMD athlon x2 2.1 GHz	2 Gb	20 Gb
Node2	AMD athlon x2 2.8 GHz	3 Gb	20 Gb
Node3	Intel PentiumR Dual 1.80 GHz	1 Gb	20 Gb
Node4	Intel Core i5 3.2 GHz	3 Gb	20 Gb

Table I
ENVIRONMENT

B. Experiments

Experiments were conducted with CyclopsDCMServer sequential and parallel architecture. The experiments involve only comparison of writing a new DICOM file in HDF file; future comparisons will compare file retrieval. This experiment measures the time spent write an image buffer into a HDF data set and was done 25 times with different DICOM files for each test. The selection of the files used was random, but the same files were used for the parallel process. The time collected is the time required to write an image, ignoring other information, like patient name,

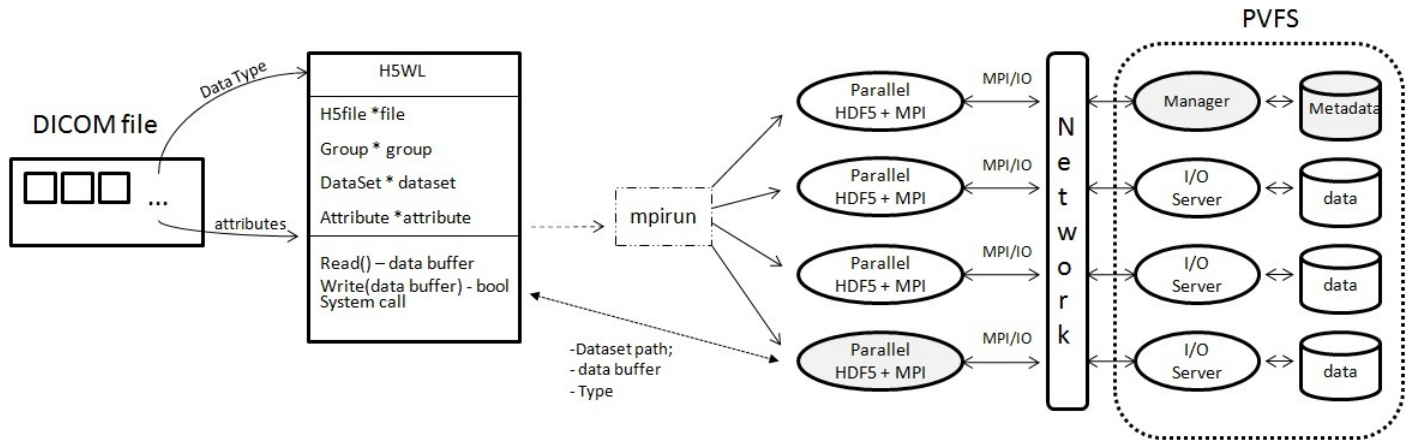


Figure 2. The architecture proposed

hospital and etc. The reason that this measure was chosen is because an image represents most of a DICOM file, i.e., could be over ninety percent and is normally nearly 50 Mb. These images were created by CT equipment that generates monochromatic images with 512 512 pixels with 16 bits per pixel.

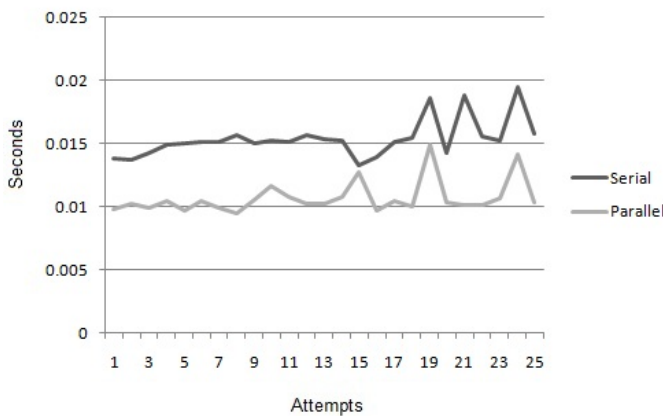


Figure 4. Contiguous Hyperlasb

Figure 4 compares the performance between serial CycloDCMServer and our integrated parallel architecture. Our architecture shows better performance than the serial with the average writing time for parallel method being 0.0107176 seconds, while the average writing time for the serial was 0.01539956 seconds an improvement of around 30 percent. The minimal elapsed times were 0.013812s and 0.009815s for serial and parallel respectively, while the maximal elapsed times were 0.019435s and 0.012735s for serial and parallel.

VIII. CONCLUSION AND FUTURE WORKS

In this paper, was introduced an extension to the architecture for the CyclopsDCMServer that was introduced by Macedo et al. [1]. The focus of this paper was to introduce a new parallel architecture designed to reduce the bottleneck I/O issues in the serial architecture. Experiments involving writing compared the serial and parallel I/O in the same environment and show that the parallel architecture resulted in a thirty percent improvement.

As future work, there is a need to compare the retrieval of a DICOM image from an HDF file using the parallel architecture. It is not clear how this will perform since this uses H5WL which will not perform as well as a standard DBMS.

Another important experiment is to measure the performance of the complete operations of receiving a DICOM file, wrapping and storing it, and the reverse operation of retrieving and unwrapping it. Others future work includes analyzing the significance of the number of MPI nodes on reading and writing and to measure the communication between H5WL and master mpi node.

As seen in the Section 6, many researchers that have similar issues have used parallel I/O to avoid bottleneck I/O problems and have obtained similar results. Given this and considering the features available in HDF, one can expect to see gains in other future work involving the retrieval of stored information.

REFERENCES

[1] D. de Macedo, H. Perantunes, L. Maia, E. Comunello, A. von Wangenheim, and M. Dantas, "An interoperability approach based on asynchronous replication among distributed internet databases," in *Computers and Communications, 2008. ISCC 2008. IEEE Symposium on*, 2008, pp. 658 –663.

- [2] W. Hersh, U. S. A. for Healthcare Research, Quality, and O. H. S. U. E. based Practice Center, *Telemedicine for the Medicare population: Update*. Citeseer, 2006.
- [3] “Laboratorio de telemedicina,” Access:, January 2011. [Online]. Available: <http://www.telemedicina.ufsc.br>
- [4] 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.
- [5] “Cyclops project,” Available at:<http://www.cyclops.ufsc.br> . Access: 2011., January 2011. [Online]. Available: <http://www.cyclops.ufsc.br>
- [6] “Cyclops group,” Access:, December 2011. [Online]. Available: <http://cyclops.telemedicina.ufsc.br>
- [7] “Pvfs,” Access:, January 2011. [Online]. Available: <http://www.pvfs.org>
- [8] O. Pianykh, *Digital Imaging and Communications in Medicine (DICOM): A practical introduction and survival guide*. Springer Verlag, 2008.
- [9] “Hdfgroup,” Available at:<http://www.hdfgroup.org> . Access: 2011., January 2011. [Online]. Available: <http://www.hdfgroup.org>
- [10] “Romio,” Access:, March 2011. [Online]. Available: <http://www.mcs.anl.gov/research/projects/romio/>
- [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. Adelman, 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] H. Yu, K. Ma, and J. Welling, “A parallel visualization pipeline for terascale earthquake simulations,” in *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*. IEEE, 2005, p. 49.
- [14] “Mpirun,” Access:, March 2011. [Online]. Available: <http://www.mcs.anl.gov/research/projects/mpi/www/www1/mpirun.html>

Dogleg Channel Routing with Parallel Mixed Integer Linear Programming Solvers

I-Lun Tseng¹, Yung-Wei Kao¹, Cheng-Yuan Chang¹, and Adam Postula²

¹Department of Computer Science and Engineering, Yuan Ze University, Taiwan

²School of Information Technology and Electrical Engineering, The University of Queensland, Australia

Abstract - Channel routing is a type of problems arising in the detailed routing phase of VLSI physical design automation as well as in the design of printed circuit boards (PCBs). It has been known that channel routing problems can be formulated as constraint programming (CP) problems and thus CP solvers can be used to find solutions. In this article, we present a mixed integer linear programming (MILP) formulation to gridded dogleg channel routing problems. As a result, parallel MILP solvers, which have the ability to exploit the computational power of multi-core processors, can be used to find solutions. Experimental results show that high degrees of scalability can be achieved. Owing to the properties of MILP problems, it is possible to further shorten the execution time if a computer containing more processor cores is available.

Keywords: Channel Routing, VLSI Physical Design Automation, Mixed Integer Linear Programming, Parallel Computing

1 Introduction

Channel routing is a type of problems arising in the detailed routing phase of VLSI physical design automation [2] as well as in the design of printed circuit boards (PCBs) [3]. Although channel routing has not been an active research field in recent years, the use of mixed integer linear programming (MILP) technologies in solving this type of problems has not been completely investigated. Moreover, gaining full understanding of these fundamental problems is essential to the research and development of other routing algorithms [4].

In order to solve a channel routing problem, many routing algorithms generate horizontal and/or vertical constraints for the problem [5, 6]. For a channel routing problem containing cyclic vertical constraints, doglegs are required in order to complete the routing [6]. Since dogleg channel routing problems are NP-complete [7], many heuristic algorithms have been developed and proposed [6, 8, 9]. Unfortunately, those heuristic algorithms are not guaranteed to generate optimal solutions.

Instead of developing heuristic routing algorithms, we transform a gridded dogleg channel routing problem into an

MILP problem. Consequently, an MILP solver can be used to find optimal solutions of the transformed problem. As a result, the number of tracks can be minimized. With this minimum number of tracks, furthermore, the number of vias can also be minimized.

A channel routing problem can be considered as a multi-objective optimization problem, as we may need to simultaneously optimize two or more objectives, such as minimizing the number of tracks [8], minimizing the number of vias [10], minimizing the crosstalk [11, 12], and minimizing the total wire length [13]. Most heuristic routing algorithms only consider one or two of those objectives, and adding other objectives may result in redesign of those algorithms. Although this article focuses on the objectives of minimizing the number of tracks and the number of vias, our approach can be further extended to consider other objectives (e.g., crosstalk minimization [11]).

This paper is organized as follows. In Section 2, we present a systematic approach which can be used to transform a constraint programming (CP) problem into an MILP problem. Section 3 discusses dogleg channel routing problems and the representations of their routing results. Left and right connection sets are presented in Section 4, followed by the MILP-based channel routing algorithm. Section 6 shows experimental results. Finally, conclusions are drawn in Section 7.

2 CP to MILP Transformation

Constraint programming (CP) [14] is a type of declarative programming paradigm since it allows users to specify a problem in terms of variables and constraints over those variables. After a problem has been specified, a CP solver can be used to find the solution(s) to the problem. CP solvers have been used in solving many difficult problems (such as optimization problems, scheduling and resource assignment problems [15, 16], and problems of partitioning parameterized polygons [17]), although the time complexity for solving those problems may not be polynomial.

Linear programming (LP) [18] also allows users to specify a problem in terms of variables and constraints. However, each of these constraints must be a linear equality or a linear inequality constraint. In addition, a linear objective function can exist and its value can be either minimized or maximized. If a problem can be expressed in this way, it is

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.

called an LP problem. If a problem can be expressed as an LP problem and all of the variables are required to be integers, then it is called an integer linear programming (ILP) problem. Furthermore, if a problem can be expressed as an LP problem and some of the variables are required to be integers, it is called a mixed integer linear programming (MILP) problem.

Over the past few years, technologies of MILP solvers have advanced. A number of commercial MILP solvers (e.g., CPLEX and Gurobi Optimizer) are now capable of exploiting computational power from multi-core processors. Therefore, by transforming a CP problem into an MILP problem, parallel MILP solvers can be used to solve the original problem. As a result, computational resources from multi-core processors can be highly utilized and the execution time can be reduced. This section describes how to transform a number of fundamental types of linear and/or logical constraints, which reside in a CP problem, into linear inequalities.

In this section, symbols A , B , C , and D are used; each of them can be a constant, a variable, or a linear expression. Also, we assume that values for all of the constants, variables, and linear expressions are integers.

2.1 Logical-AND Constraints

Since a CP problem can have many constraints and a solution to the CP problem must satisfy all of the constraints, the relation between each pair of these constraints is considered as conjunction (logical-AND). In a CP solver, therefore, when we need to specify a set of conjunctive constraints, usually we only need to impose these constraints one by one. Similarly, in an MILP solver, conjunctive constraints can be specified one by one.

2.2 Equality Constraints

For a constraint whose type is in the form of $(A = B)$, it can be transformed into two conjunctive inequalities $(A \leq B)$ and $(B \leq A)$; each of A and B can be a constant, a variable, or a linear expression. As an example, the linear constraint $(3*A - 2*B = 5*C + 26)$ can be transformed into two linear inequalities.

2.3 Less-Than and Greater-Than Constraints

A constraint which is in the form of $(A < B)$ can be converted into the inequality $(A \leq B - 1)$ if values of both A and B are integers. Likewise, a constraint which is in the form of $(A > B)$ can be transformed into the inequality $(B \leq A - 1)$.

2.4 Logical-OR Constraints

In order to convert a constraint which contains a logical-OR operation into inequalities, an extra binary variable, whose value is either 0 or 1, can be used; the binary variable can also be called a 0-1 integer. For example, a constraint which is in the form of $((A \leq B) \text{ OR } (C \leq D))$ can be transformed into the following two conjunctive inequalities:

- $A \leq B + M1*BV$
- $C \leq D + M2*(1-BV)$

In the above inequalities, BV is a binary variable. Also, both of $M1$ and $M2$ are constants whose values must be sufficiently large so that the second inequality is redundant if $BV=0$ and the first inequality is redundant if $BV=1$. Similarly, a constraint which contains two or more logical-OR operations can be converted into a number of inequalities by using two or more binary variables, as one of the examples shown in Section 2.6.

2.5 Not-equal Constraints

A not-equal constraint which is in the form of $(A \neq B)$ can be converted into two conjunctive inequalities as the steps shown below:

$$\begin{aligned} A \neq B & \\ \Leftrightarrow (A < B) \text{ OR } (B < A) & \\ \Leftrightarrow (A \leq B - 1) \text{ OR } (B \leq A - 1) & \quad [\text{Section 2.3}] \\ \Leftrightarrow (A \leq B - 1 + M1*BV) \text{ AND } (B \leq A - 1 + M2*(1-BV)) & \quad [\text{Section 2.4}] \end{aligned}$$

As described in Section 2.4, BV is a binary variable, and the values of $M1$ and $M2$ must be sufficiently large.

2.6 If-Then-Else and If-Then Constraints

This subsection shows two examples of conditional constraints and how they can be transformed into linear inequalities. Other types of conditional constraints can be transformed by using similar techniques. The first example is the following If-Then-Else constraint:

- IF $(A \leq B)$ THEN $(C \leq D)$ ELSE $(E \leq F)$

The constraint can be split into two disjunctive (logical-OR) constraints as shown below:

- $((A \leq B) \text{ AND } (C \leq D))$
- $((A > B) \text{ AND } (E \leq F))$

We can then add a binary variable, BV , in order to further transform these two disjunctive constraints. The final transformed conjunctive inequalities are listed below:

- $A \leq B + M1*BV$
- $C \leq D + M2*BV$
- $B \leq A - 1 + M3*(1-BV)$
- $E \leq F + M4*(1-BV)$

All of $M1$, $M2$, $M3$, and $M4$ are integer constants and their values must be sufficiently large. Therefore, $(A \leq B)$ and $(C \leq D)$ must hold when $BV=0$. Also, $(A > B)$ and $(E \leq F)$ must hold when $BV=1$.

The second example is the following If-Then constraint, which is more complicated than the previous example but can also be converted into linear inequalities:

- IF $(A < B)$ THEN $(C = D)$ OR $(E < F \text{ AND } G < H)$ OR $(I < J \text{ AND } K < L)$

Table I. Conjunctive Inequalities Transformed from the Constraint "IF ($A < B$) THEN ($C = D$) OR ($E < F$ AND $G < H$) OR ($I < J$ AND $K < L$)"

Constraint ID	Constraint	Values of Binary Variables for Activating Corresponding Constraints
C-1	$A \leq B - 1 + M1*B1 + M2*B2$	If ($B1 = 0$) and ($B2 = 0$), then ($A < B$) and ($C = D$).
C-2	$C \leq D + M3*B1 + M4*B2$	
C-3	$D \leq C + M5*B1 + M6*B2$	
C-4	$A \leq B - 1 + M7*B1 + M8*(1-B2)$	If ($B1 = 0$) and ($B2 = 1$), then ($A < B$) and ($E < F$) and ($G < H$).
C-5	$E \leq F - 1 + M9*B1 + M10*(1-B2)$	
C-6	$G \leq H - 1 + M11*B1 + M12*(1-B2)$	
C-7	$A \leq B - 1 + M13*(1-B1) + M14*B2$	If ($B1 = 1$) and ($B2 = 0$), then ($A < B$) and ($I < J$) and ($K < L$).
C-8	$I \leq J - 1 + M15*(1-B1) + M16*B2$	
C-9	$K \leq L - 1 + M17*(1-B1) + M18*B2$	
C-10	$B \leq A + M19*(1-B1) + M20*(1-B2)$	If ($B1 = 1$) and ($B2 = 1$), then ($A \geq B$).

This constraint can be divided into four disjunctive (logical-OR) constraints as listed below:

- ($A < B$) AND ($C = D$)
- ($A < B$) AND ($E < F$) AND ($G < H$)
- ($A < B$) AND ($I < J$) AND ($K < L$)
- ($A \geq B$)

We can then add two binary variables, $B1$ and $B2$, in order to further transform these disjunctive constraints. The transformation result, which contains ten conjunctive inequalities, is shown in Table I.

3 Problem Formulation

In a channel routing problem, a channel is a rectangular region bounded by two parallel rows (the top row and the bottom row). The two parallel rows have terminals and each terminal has a number, which represents the name of a net. Terminals having the same number must be connected together, except that terminals with the number zero require no connection.

In this paper, it is assumed that a channel routing problem has only two routing layers, one layer for horizontal wire segments and the other for vertical wire segments. Endpoints of wire segments must be located within the channel (the rectangular region). For the wire segments that reside on different layers, in addition, they can be connected by *vias*. In the figures in this paper, vias are denoted by small black squares.

Figure 1 shows an example of a channel routing problem and one of its solutions. The problem has six columns and each terminal lies at the intersection of a row and a column. Moreover, horizontal wire segments, which are used for routing purposes, must lie on the tracks. As can be seen in this example, the routing solution uses three tracks. The columns, rows, and tracks form an array of (virtual) grids. Therefore, the channel routing problem shown in Figure 1 is a *gridded* channel routing problem if all the endpoints of (horizontal and vertical) wire segments are restricted to lie on the grids.

In a channel routing problem, since the width of the channel is fixed, minimizing the routing area is equivalent to

minimizing the number of tracks (the height of the channel). By introducing doglegs [19] in solving the problem shown in Figure 1, it is possible to complete the routing with only two tracks, as shown in Figure 2. The use of doglegs in solving channel routing problems is a technique of great importance. In the cases where cyclic vertical constraints exist [20], doglegs must be used in order to complete the routing.

In our model of a gridded dogleg channel routing problem, each net is composed of a number of horizontal and vertical wire segments. In addition, these horizontal wire segments must be placed between the net's leftmost column and rightmost column. For the channel routing problem given in Figure 1, the horizontal span of each net is shown in Figure 3. Based on the horizontal spans, a number of horizontal wire *fragments* (or smaller horizontal wire segments), as shown in Figure 4, can be generated by cutting the horizontal spans into pieces. Each of these horizontal wire fragments spans between two adjacent columns. In addition, the union of all the horizontal wire fragments of one net must cover the total horizontal span of the net. The name of each horizontal wire fragment is coded as follows (as the example shown in Figure 4):

$\langle \text{net name} \rangle @ \langle \text{left column no.} \rangle _ \langle \text{right column no.} \rangle$

In our model of a channel routing problem, each horizontal wire fragment is associated with a numerical value; the value represents the track on which the wire fragment is located. In our algorithm, moreover, vertical wire segments are not cut into fragments; positions of vertical wire segments can be decided easily after all the horizontal wire fragments have been placed. For instance, the routing result shown in Figure 2 can be represented by the following code:

[1@2_3=1, 1@3_4=2, 1@4_5=2, 1@5_6=2, 2@1_2=2, 3@4_5=1, 3@5_6=1]

4 Left and Right Connection Sets

A channel routing problem can have a left connection set (LCS) and/or a right connection set (RCS) [4]. The left (right) connection set refers to the set of nets which enters/exits the channel from the left (right). To tackle these types of nets, we

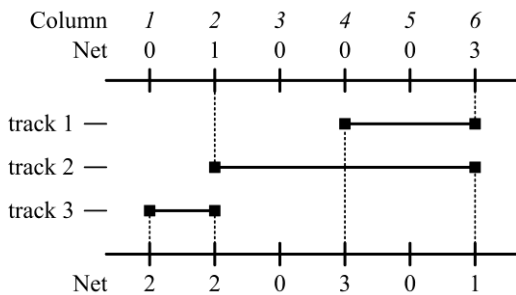


Figure 1. A (gridded) channel routing problem and one of its solutions without using doglegs

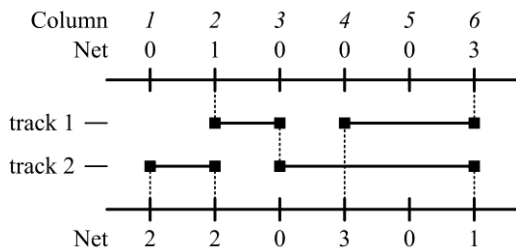


Figure 2. A solution to the channel routing problem (shown in Figure 1) with the use of doglegs

add a new column at the left of the channel if the LCS exists, and add a new column at the right of the channel if the RCS exists. Also, new horizontal fragments must be created. For the newly created columns, their corresponding vertical constraints [21] do not need to be generated. However, horizontal constraints [21] for the newly created fragments are required. Figure 5 illustrates an example of a channel routing problem with the existence of LCS and RCS; the problem can be transformed into the one shown in Figure 6.

5 The Algorithm

The methodology of formulating gridded dogleg channel routing problems as CP problems has been presented in [21]. Additionally, by using the methodology presented in Section 2, horizontal and vertical constraints presented in [21] can be transformed into conjunctive linear inequalities. As a result, a dogleg channel routing problem formulated as a CP problem can be transformed into an MILP problem. The final MILP-based dogleg channel routing algorithm is shown in Figure 7. To the best of our knowledge, our approach is the first in the literature to model the problems of gridded dogleg channel routing with via minimization as MILP problems. Moreover, our approach does not require a given initial routing result as the input.

In our MILP-based channel routing algorithm, a gridded dogleg channel routing problem is transformed into an MILP problem and then solved by an MILP solver. Since *channel density* is the minimum number of tracks required in order to solve a two-layer channel routing problem [9], our algorithm uses it as the initial value for the number of available tracks.

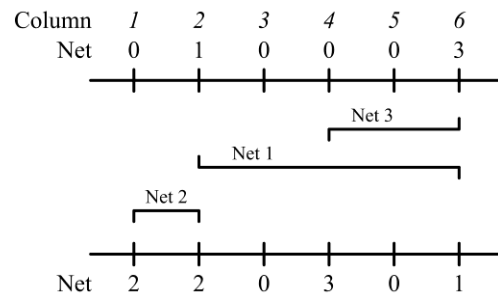


Figure 3. Horizontal span of each net

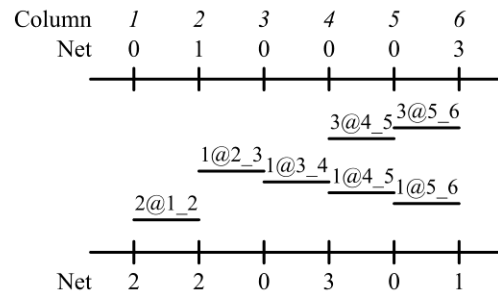


Figure 4. Horizontal wire fragments and their representations

Therefore, the domain for the position of each horizontal wire fragment is set to $[1, \text{channel density}]$ initially. In the algorithm, moreover, if the transformed MILP problem cannot be solved by using the specified number of available tracks, the algorithm will increase the number of available tracks by 1 and then solve the transformed problem again; it is also possible to override the value of *Max* manually. The algorithm stops when a solution has been found.

6 Experimental Results

The proposed algorithm has been implemented in Java programming language. Also, a number of testcases have been used to verify the correctness and to measure the performance of our MILP-based channel routing program. Information for some of those testcases is shown in Table II, and the experimental results are shown in Table III. Note that Table III shows results of executing our programs with the via minimization function turned on. Testcases and routing results of “Figure 2” and “Figure 9” can be seen in Figure 2 and Figure 9, respectively. Figures 8 and 9 illustrate the same routing problem with different routing results; one is with the via minimization function turned off and the other turned on. Testcases “Phillips_14” and “Phillips_16” are from [1]; the former has 14 columns and the latter has 16 columns. The testcase named “Deutsch” is the notable Deutsch’s difficult example [4, 19].

To compare the efficiency of the MILP-based channel routing program with the CP-based channel routing program [21], we used JaCoP version 2.4.1 (released in 2009) as the CP solver. In our MILP-based program, we could choose to

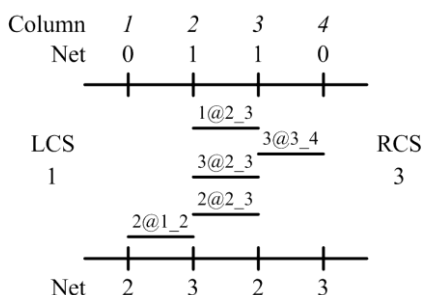


Figure 5. A channel routing problem with the existence of LCS and RCS

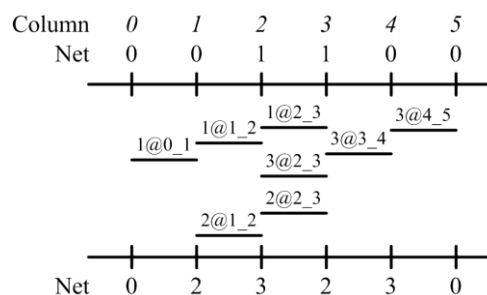


Figure 6. A new channel routing problem transformed from the channel routing problem shown in Figure 5

use CPLEX version 10.2 (released in 2007), Gurobi version 2.0.2 (released in 2009), or CPLEX version 12.1 (released in 2009) as the MILP solver. In our experiments, all of the CP and MILP solvers were run on a workstation which had an Intel Q9550 CPU (2.83 GHz, Quad Core) and 8 GB of RAM, except that CPLEX v.10.2, which was run on a server having an Intel Xeon E5345 CPU (2.33 GHz, Quad Core) and 32 GB of RAM. Also, we extracted some experimental results from [1] and put them in Table III; a machine with a 12 MHz CPU was used to generate the original results. Please note that Phillips did not implement the via minimization function.

In [1], the Phillips_16 case with 8 tracks could not be solved within 40 hours of execution time and no result was generated. By using our MILP-based channel routing algorithm with CPLEX v.12.1, we proved that the Phillips_16 case with 8 tracks was infeasible within one minute of time. From the experimental results shown in Table III, we found that CPLEX v.12.1 performed efficiently in identifying infeasible problems, while Gurobi v.2.0.2 performed relatively well in solving large and complex feasible problems. In addition, the performance of JaCoP v.2.4.1 was relatively unstable, but it solved some large problems rapidly.

Although our approach of using MILP solvers is capable of finding optimal solutions, the execution time can be very long. That is because dogleg channel routing problems are NP-complete. However, as can be seen in the Deutsch case in Table III, suboptimal results can be generated in exchange for significantly reduced execution time. Moreover, as will be shown in the next paragraph, if more processor cores are available, the execution time can be reduced.

Table IV shows the elapsed time of running MILP-based channel routing with different parallel MILP solvers and with different number of threads. While we were running testcases by using four threads, it was common to see that the CPU utilization reached around 400% (by using Linux's "top" command), which meant that all of the four CPU cores had participated in the computation. In general, as shown in Table IV, if more CPU cores are available, the execution time can be reduced. In the results of solving the Deutsch's difficult problem with 25 tracks, it was interesting to note that nearly 8x speedup had been achieved when the number of threads was increased from one to four. That is because, while using a parallel solver in solving an optimization problem, one thread can pass information to other threads when a feasible solution

Algorithm MILP-CHANNELROUTER(TR, BR, LCS, RCS, VM)

Input. The input contains the description of a gridded dogleg channel routing problem, which includes (1) TR , which is the list of terminals at the top row, (2) BR , which is the list of terminals at the bottom row, (3) LCS , and (4) RCS . Also, a Boolean variable VM is used in order to control the activation of the via minimization function.

Output. The output contains a permutation of horizontal wire fragments, from which a solution to the input channel routing problem can be constructed.

1. $D \leftarrow$ the channel density of the input channel routing problem
2. $Max \leftarrow D$
3. **Do** {
4. Generate variables for horizontal wire fragments; the domain of each variable is set to $[1, Max]$.
5. Generate horizontal constraints for each column interval.
6. Generate vertical constraints for each column.
7. Generate constraints for the left and right connection sets (Section 4).
8. **if** VM equals TRUE **then**
9. Generate variables and constraints for minimizing the number of vias.
10. Specify generated variables and constraints via API functions of an MILP solver.
11. Invoke the MILP solver to solve the specified MILP problem. When a solution has been found, report the solution and then exit the algorithm.
12. $Max \leftarrow Max + 1$
13. } **while** (the MILP solver has not found a solution)

Figure 7. The MILP-based dogleg channel routing algorithm

has been found by the thread. As a result, many unnecessary computations can be eliminated.

7 Conclusion

In order to exploit the computational power of multi-core processors, we proposed a systematic approach which can be used to transform a CP problem containing linear and logical constraints into an MILP problem. The approach is capable of transforming a gridded dogleg channel routing problem with via minimization, formulated as a CP problem, into an MILP problem. Experimental results have shown that high degrees

Table II. Information of Channel Routing Problems

Testcase	# of nets	# of columns	channel density	min. # of tracks
Figure 2	3	6	2	2
Figure 9	5	9	5	6
Phillips_14	10	14	4	5
Phillips_16	11	16	8	9
Deutsch	52	156	19	19

of nets: the number of nets in the testcase; # of columns: the number of columns in the testcase; channel density: the channel density of the testcase; min. # of tracks: the minimum number of tracks required for solving the channel routing problem.

Table III. Experimental Results of CP-based and MILP-based Channel Routing

Testcase	# of tracks used	Phillips [1]	JaCoP v.2.4.1 1 thread	CPLEX v.10.2 1 thread	Gurobi v.2.0.2 4 threads	CPLEX v.12.1 4 threads
Figure 2	2	N.A.	0.14 sec.	0.00 sec.	0.00 sec.	0.00 sec.
Figure 9	6	N.A.	0.37 sec.	1.96 sec.	0.48 sec.	0.81 sec.
	5	N.A.	154766 sec. (infeasible)	0.24 sec. (infeasible)	0.32 sec. (infeasible)	0.21 sec. (infeasible)
Phillips_14	6	0.28 sec.	253.55 sec.	0.32 sec.	0.27 sec.	0.20 sec.
	5	< 1.37 sec.	0.36 sec.	0.08 sec.	0.04 sec.	0.09 sec.
	4	< 1.37 sec. (infeasible)	0.19 sec. (infeasible)	0.00 sec. (infeasible)	0.00 sec. (infeasible)	0.00 sec. (infeasible)
Phillips_16	10	N.A.	> 7 days (unfinished)	8.49 sec.	0.71 sec.	0.92 sec.
	9	0.39 sec.	> 7 days (unfinished)	27.37 sec.	0.67 sec.	0.93 sec.
	8	> 40 hrs. (unfinished)	> 7 days (unfinished)	54646.89 sec. (infeasible)	520.23 sec. (infeasible)	55.34 sec. (infeasible)
Deutsch	25	N.A.	4.18 sec.	> 3 days (unfinished)	3451.79 sec.	> 7 days (unfinished)
	24	N.A.	5.10 sec.	> 3 days (unfinished)	174270.72 sec.	> 7 days (unfinished)
	23	N.A.	> 7 days (unfinished)	> 3 days (unfinished)	136266.61 sec.	> 7 days (unfinished)

of tracks used: the number of tracks used by our CP-based or MILP-based channel routing program for solving the problem (the value was assigned by setting the *Max* manually).

of scalability can be achieved by using parallel MILP solvers. Our approach can be further extended to consider crosstalk and total wire length. Although the experimental results show that the execution time of our approach cannot compete with many existing channel routers, optimal results can be generated for small to medium cases. In addition, for large cases, suboptimal results can be generated in exchange for significantly reduced running time. We believe that the performance of our approach can be further improved with the advance of mixed integer linear programming technologies as well as the advance of multi-core processors.

References

- [1] Nicholas C. Phillips, "Channel Routing by Constraint Logic," In *Proceedings of ACM Symposium on Applied Computing*, pp. 536-540, 1992.
- [2] Naveed A. Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers, 1999.
- [3] Akihiro Hashimoto, and James Stevens, "Wire Routing by Optimizing Channel Assignment within Large Apertures," In *Proceedings of Design Automation Conference*, pp. 155-169, 1971.
- [4] Rajat K. Pal, *Multi-Layer Channel Routing: Complexity and Algorithms*, Narosa Publishing House, 2000.
- [5] Jia-Shung Wang, and R. C. T. Lee, "An Efficient Channel Routing Algorithm to Yield an Optimal Solution," *IEEE Transactions on Computers*, vol. 39, no. 7, pp. 957-962, July, 1990.
- [6] Takeshi Yoshimura, and Ernest S. Kuh, "Efficient Algorithms for Channel Routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-1, no. 1, pp. 25-35, January, 1982.
- [7] Thomas G. Szymanski, "Dogleg Channel Routing is NP-Complete," *IEEE Transactions on Computer-Aided Design*, vol. CAD-4, no. 1, pp. 31-41, 1985.
- [8] James Reed, Alberto Sangiovanni-Vincentelli, and Mauro Santomauro, "A New Symbolic Channel Router: YACR2," *IEEE Transactions on Computer-Aided Design*, vol. CAD-4, no. 3, pp. 208-219, 1985.
- [9] Uzi Yoeli, "A Robust Channel Router," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 2, pp. 212-219, February, 1991.
- [10] Chung-Kuan Cheng, and David N. Deutsch, "Improved Channel Routing by Via Minimization and Shifting," In *Proceedings of Design Automation Conference*, pp. 677-680, 1988.
- [11] Tong Gao, and C. L. Liu, "Minimum Crosstalk Channel Routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 5, pp. 465-474, May, 1996.

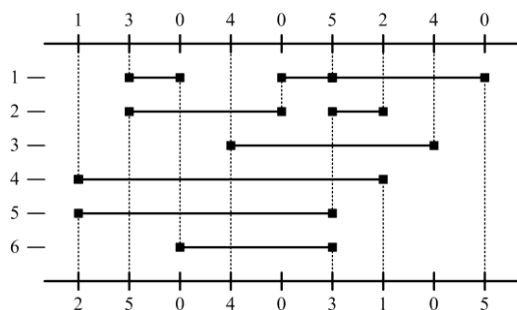


Figure 8. A minimum-track solution to a dogleg channel routing problem without via minimization (total number of vias = 17)

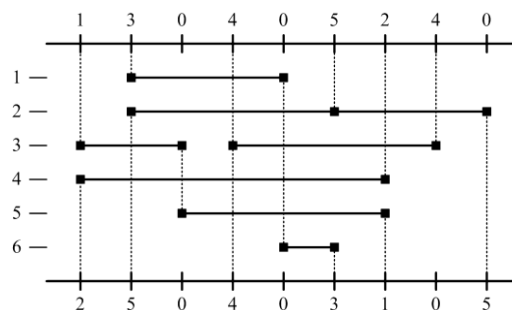


Figure 9. A minimum-track solution to the dogleg channel routing problem (shown in Figure 8) with the via minimization function turned on (total number of vias = 15)

Table IV. Executing MILP-Based Channel Routing with Different Number of Threads

Testcase	# of tracks used	CPLEX v.12.1			Gurobi v.2.0.2	
		1 thread	2 threads	4 threads	1 thread	4 threads
Phillips_16	8	209.42 sec.	127.87 sec.	55.34 sec.	1708.20 sec.	520.23 sec.
Deutsch	25	> 3 days (unfinished)	> 3 days (unfinished)	> 3 days (unfinished)	27410.08 sec.	3451.79 sec.

- [12] Kuo-Chih Hsu, Yu-Chung Lin, Po-Xun Chiu, and Tsai-Ming Hsieh, "Minimum Crosstalk Channel Routing with Dogleg," In *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 73-76, 2000.
- [13] Pralay Mitra, Nabin Ghoshal, and Rajat K. Pal, "A Graph Theoretic Approach to Minimize Total Wire Length in Channel Routing," In *Proceedings of IEEE Region 10 Conference (TENCON)*, pp. 414-418, 2003.
- [14] Kim Marriott, and Peter J. Stuckey, *Programming with Constraints: An Introduction*, The MIT Press, 1998.
- [15] Krzysztof Kuchcinski, "Constraints-Driven Scheduling and Resource Assignment," *ACM Transactions on Design Automation of Electronic Systems*, vol. 8, no. 3, pp. 355-383, 2003.
- [16] Krzysztof Kuchcinski, and Christophe Wolinski, "Global Approach to Assignment and Scheduling of Complex Behaviors Based on HCDG and Constraint Programming," *Journal of Systems Architecture*, vol. 49, pp. 489-503, 2003.
- [17] I-Lun Tseng, and Adam Postula, "Partitioning Parameterized 45-Degree Polygons with Constraint Programming," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, no. 3, pp. 52:1-52:29, July, 2008.
- [18] Wayne L. Winston, and Munirpallam Venkataramanan, *Introduction to Mathematical Programming*, Thomson Learning, Inc., 2003.
- [19] David N. Deutsch, "A 'Dogleg' Channel Router," In *Proceedings of Design Automation Conference*, pp. 425-433, 1976.
- [20] Takeshi Yoshimura, and Ernest S. Kuh, "Efficient Algorithms for Channel Routing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-1, no. 1, pp. 25-35, January 1982.
- [21] I-Lun Tseng, Huan-Wen Chen, Che-I Lee, and Adam Postula, "Constraint-Based Dogleg Channel Routing with Via Minimization," In *Proceedings of International Conference on Artificial Intelligence (ICAI)*, pp. 666-672, 2010.

Thick Control Flows: Introduction and Prospects

Ville Leppänen^α, Martti Forsell^β, Jari-Matti Mäkelä^α

^αDepartment of Information Technology, University of Turku, Finland,

Email: { Ville.Leppanen, jmjak } @utu.fi

^βVTT, Platform Architectures, Oulu, Finland; Email: Martti.Forsell@vtt.fi

Abstract—*Besides correctness of programming, the technical software problem of many contemporary approaches to multithreaded computations is how to organize the co-operation of a huge number of parallel threads. For the software problems, we propose parallel thick control flows as the solution.*

*When a thick control flow (in terms of the number of threads) is executing a statement or an expression of a program, all the threads are considered to execute the same program element synchronously in parallel. Considering method calls, when a control flow with thickness t calls a method, the method is **not** called separately by each of the t threads, but the control flow calls it only once with t threads. A call stack is not related to each thread but to each of the parallel control flows, since threads do not have program counters – only control flows have program counters. The concept of thread is only implicit. A thick thread-wise variable is an array-like value having a thread-wise actual value. Method signatures naturally advance types with thickness, but non-thick types are also useful.*

The concept of thick control flow makes the programmer to focus on co-operation of few parallel thick control flows instead of a huge number of parallel threads. The concept computation's state is promoted as a flow is seen to have a state (instead of each thread). The concept of state has been in a central role in achieving correctness in sequential programs. The concept of thick control flows is related to data parallelism and stream computing. It is a natural generalization of ordinary imperative sequential programming.

1. Introduction

Processor manufacturers provide multicore solutions where the cores run threads. Threads as a concept have a long history in programming languages as well as in the operation of processor cores. We argue that perhaps the thread as a concept should remain a concept for the operation of processor cores but a more abstract

concept of *thick control flow* should be used instead of thread in programming languages. Moreover, some implementation choices for thick control flows suggest that a concept replacing the thread concept could also be useful for the processor cores, since ultimately processor cores should support a very flexible number of parallel “threads” and achieving the flexibility might require a different concept.

Consider multithreaded programs. Each thread runs along a control flow specified by the program in form a method/procedure as in e.g. Java and Cilk [4]. At hardware level it may be natural to consider there to exist hundreds or even thousands of threads each running along a separate control flow, but when designing a program, it is hardly feasible to consider describing thousands of different kinds of threads. Typically the thousands of threads of a parallel program run in groups following a few control paths. Our concept of thick control flows simply groups threads more tightly into each other.

A software engineering perspective aiming at (reasoning about) correctness of parallel programs provides another fact supporting grouping of threads more tightly to each other and making each group to proceed synchronously. Reasoning about correctness of sequential programs or testing their execution has been successful, as one has been able to see the execution of a program as a sequence of state transitions. In this respect, synchrony is a language-level or library-level mechanism to define states. The more the threads can do operations between states, the harder it will be to reason about the correctness of state transitions. If a huge number of threads can proceed at arbitrary speed, the program can be in very many different states. Such multithread programming is very difficult in correctness sense. Having only a few thick parallel synchronous control flows in the program makes it much easier to reason about the correctness, since such a mode of execution significantly reduces the amount

of possible program states.

In this paper we propose a new concept of thick control flows for defining parallel (multithreaded) programs, and consider its influence on the semantics of ordinary language constructions. Next, in Section 2 we elaborate further the thick control flow idea. Section 3 discusses some example programs written along the idea, although the idea of this paper is not to propose any specific language syntax for the constructions discussed in Section 2, rather the presented constructions can be seen to extend several common imperative languages. In Section 4, we discuss related work and consider architectural requirements supporting thick control flows and speculate with possibilities to reduce the need of architectural support via compiling techniques. In fact, it will be rather straightforward to translate thick control flow based programs as ordinary sequential programs. Finally in Section 5, we draw conclusions.

2. Thick control flows

The concept of thick control flows is a straightforward generalization of ordinary sequential program flows. A typical sequential program implicitly defines a lots of different kinds of control paths “through” the program. When a sequential program is run with a single thread, the thread follows exactly one of the possible control paths. From a semantical viewpoint, one can consider that even a single-threaded program runs through all of its possible control paths in parallel. At each program statement involving a conditional branching of control (if-statements, switch-statements, loop condition checks, ...), the control of a single-threaded control flow can be considered to advance to all possible branches with thickness either one or zero threads – naturally we must require that the incoming thickness matches with the sum of outgoing thicknesses. Executing a control flow of zero thickness naturally has no effect (and can be ignored in practice).

2.1 Basic idea

When a thick control flow (in terms of the number of threads) is executing a statement or an expression of a program, all the threads are considered to execute the same program element in parallel. More precisely, we consider that some of the program variables can be replicated thread-wise – conceptually meaning that there is a unique thread-wise instance of the variable.

An expression based on using a replicated variable is conceptually also replicated – meaning that it needs to be evaluated separately for each thread. Similarly, statements can also be replicated. However, all expressions or statements executed by a thick control flow do not need to be replicated – such non-replicated (ordinary) program elements naturally only need to be evaluated/executed once by the thick flow, not thread-wise. Thus, when a thick flow proceeds over a program fragment, some of the expressions/statements translate to single instructions whereas replicated ones translate to sets of instructions.

Considering method calls, when a control flow with thickness t calls a method, the method is not called separately with each thread, but the control flow calls it only once with t threads. A call stack is not related to each thread but to each of the parallel control flows. Executing a branching statement can mean temporarily splitting a thick control flow into several other flows.

Considering semantics, the concept of thick control flows can be very helpful. All threads of a control flow can be seen to synchronously march through the (common) program code. When a flow is split into separate flows, we consider that nothing is assumed about the advancing speed of the split flows. However, joining of different parts involves an implicit synchronization of the joined flows. In this respect, the concept of thick control flow promotes the concept computation's state, which has a central role in achieving correctness.

2.2 Thick flow, replicated variables and stack

Originally, a program is considered to have a flow of thickness 1, measured conceptually in number of parallel threads. A method can be considered to have a thickness related to the calling flow's thickness. For creating a thick flow, we consider having two options: Either have a statement to dynamically set the thickness of the flow (by increasing/decreasing it), or have a block statement defining that the statements of the block are executed with a given thickness. We choose to support the latter option, *thick block*, as it is more naturally related to “thick”, replicated variable declarations as well as the common idea of program stack.

A thick control flow of thickness t consists of t implicit threads. We assume them to have a unique identity expressed as an integer. Basically, we consider the identities to be between $0 \dots t - 1$, but it might be

useful to have language constructions supporting other kind of indexing of implicit threads.

By a *replicated variable*, we mean a specifically declared variable within a thick block that has a *unique instance for each thread of the flow*. The degree of replication is dynamic, depending on the actual thickness of the flow. In practice, a replicated variable can be implemented as an array allocated from the memory.

We consider that replicated variables of any type can be declared (depending on the base language). Besides replicated variables, it will be useful to be able to declare *non-replicated* ordinary variables which have only one instance per a flow executing such a declaration.

Although a replicated variable is thread-wise replicated, we do not limit its usage for the implicit thread in question. Rather, we advance an expression (e.g. the '@' symbol) for implicit thread's identity. It might be wise to support other kind of referencing mechanisms too, but we do not elaborate that issue in this paper. Besides thread identity references, we also assume to have a special expression for the (dynamic) thickness of the flow (e.g. the '#' symbol).

The thick control flow concept clearly needs a stack-like structure for handling nested calls and nested thick blocks. However, as a flow can split into separate flows, the stack concept expands to a uniquely rooted tree-structure that we call *stack tree*. Efficient implementation of stack tree is a challenge that will not be discussed in this paper.

Nesting thick and ordinary block statements is meaningful and supported. Consider a situation where a thick block B_{out} of thickness t_{out} contains an inner thick block B_{in} of thickness t_{in} . A nested block is not executed thread-wise but flow-wise, and therefore considering the flow thickness, a flow executing the inner thick block has thickness t_{in} (instead of $t_{out} \times t_{in}$). Besides efficient stack tree implementation, the only real problem related to nested thick blocks is the visibility and interpretation of variables defined in an outer block for its inner blocks. If e.g. $t_{in} > t_{out}$ and B_{out} has defined a replicated variable v , then in B_{in} there would not be a replica of v for each implicit thread of B_{in} . There are several approaches to solve this issue: E.g. (a) automatic replication of v for the $t_{in} - t_{out}$ implicit threads; (b) it is a runtime error to refer to v in B_{in} , if $t_{in} > t_{out}$; (c) replicated expression referring to v only has thickness t_{out} ; and (d) variables

of outer blocks are not visible in inner blocks. The option (d) does not follow the standard approach taken in case of ordinary nested block statements. The option (c) would lead to issues dealing with operations for replicated expressions of different thickness. As option (a) is not semantically very sensible (what initial value to use for the newly replicated instances of v), we choose to follow (b). Our choice means that a replicated expression referring to v in B_{in} refers only to $v\{0\}, v\{1\}, \dots, v\{t_{in} - 1\}$, yet we allow explicit references also to $v\{t_{in}\}, \dots, v\{t_{out} - 1\}$.

2.3 Replicated expressions, assignments and statements

Consider an expression $e ::= e_1 \oplus e_2$, where \oplus is some arithmetic-logical operator supported by the hardware. Evaluating e with a flow of thickness t , a replicated variable v as either e_1 or e_2 means the expression is replicated t times using $v\{0\}, \dots, v\{t - 1\}$ as v . If v is originally declared for a thickness less than t , a runtime error will rise. All replicated expression within a block of thickness t have the same thickness t . It is not possible to have an operator (or a function call) to have replicated expressions of different thickness as arguments/operands.

However, notice that if e is a function call having replicated expressions as arguments, then e as such is not replicated as will be explained in Section 2.5. The result of the evaluation of e can be a replicated value. In this sense, the library functions and (e.g. SIMD-style) hardware supported operators will be treated differently.

A replicated value should be seen as an array of t values, although it might not be implemented as an array when the program is executed.

Operators can require some of the operands to be non-replicated, but e.g. ordinary binary arithmetic operators have replicated operands. It is always possible to use a non-replicated expression as a replicated operand, such a replication takes place implicitly. Notice that considering execution, the value of such an expression needs to be evaluated only once. Operators supporting replicated operands typically yield replicated values, but non-replicated results are also possible (e.g. sum of all replicas).

Consider an assignment statement

$$v := e;$$

The assignment has the semantics that $v\{i\}$ will receive value $e\{i\}$ for $0 \leq i < t$. There will be no additional requirements concerning thickness of replication, since both e and v are expressions. As usual, v can be any replicated expression having a valid L-value.

Considering replication, ordinary statements are not replicated. The implicit threads do not really execute program statements as only the control flows have program counters (PCs). The implicit PC of an implicit thread receives its value from the PC of control flow.

2.4 Control structures

Consider a programming language constructs for branching statements like, `if-then-else`, `switch`, and `case`. A control flow is executing each statement or expression of a program instead of a thread. When executing a branching statement with incoming control flow thickness of t_{in} and there are k possible branches with thicknesses $t_{out}^1, \dots, t_{out}^k$, we simply require that $t_{in} = t_{out}^1 + \dots + t_{out}^k$. Statements also involve joining the splitted control flows as the branches join – our thick control flow approach assumes that there will always be an implicit join at the end of a branching statement.

Considering semantics, the concept of thick control flows can be very helpful. All threads of a control flow can be seen to synchronously march through the (common) program code. When a flow is split into separate flows, it is perhaps best to consider that nothing is assumed about the advancing speed of the split flows. However, joining of different parts involves an implicit synchronization of the joined flows.

The implicit thread identifiers form a slight problem when a flow is split into several subflows. The question is should the flow's implicit thread id's always form a continuous interval starting from zero. As a subflow can call a function (where such a property is important), here we assume that the i 'th subflow always has thread id's between $0, \dots, t_{out}^i - 1$ but there will be possibility to refer to the surrounding flow's original id-numbers too. Notice that such temporary numbering can be calculated rather efficiently with multiprefix operations and that such a calculation is not always required (depends on the id-number references present in the program code).

Loop statements are executed by the flow, and as their exit condition may involve replicated variables or references to implicit thread id-numbers, the loop

statements must also be seen as potential branching statements. Otherwise (nested) loops work as in ordinary imperative languages.

A pardo-loop of e.g. the Fork language is a block statement, which in the beginning of the block increases the thickness and at the end decreases the thickness back to the original thickness. Thus, our previously discussed thick block concept replaces the need for a pardo-loop.

Naturally, it would be possible to consider other kind of explicit and implicit ways to enhance the flow thickness besides the thick block construct.

2.5 Function calls

Regardless of the base language, we call all function/procedure/method calls as function calls. Consider function calls, when a control flow with thickness t calls a function, the function is not called separately with each thread, but the control flow calls it only once with t implicit threads.

A function declaration needs to declare if it is meaningful to call a function with a thick flow (with e.g. a keyword 'rep' in the function signature). Recall that within a function one can change the thickness by using a thick block definition. The implicit flow thickness parameter can be seen as a possibility for the caller to define how many implicit threads should be used inside the function. For thick functions, it is meaningful to tie the thickness of passed values and return value to the thickness of the calling flow. Notice that it will still be useful to be able to define non-replicated types in a thick function's signature.

If the return value of a function is of replicated type, a normal return of a thick function call can be seen to return an array of values. It is also possible to consider some of the implicit threads to end their execution to an exception – then the thickness of normally returning control flow can be seen to reduce when some fraction of the incoming control flow has been separated by exceptions. The normal exception handling mechanisms (of e.g. OO languages) can be seen to later join the separated parts of the control flow with the original control flow.

In general, there are quite a lot of function signature possibilities, e.g. a function could have a specification that the return value from the call has twice the thickness of function call. However, having the outgoing thickness as twice that of the incoming thickness is not

seen among the options. We do not further elaborate with the issue in this paper.

2.6 Object-oriented features

Due to space limitations, we do not elaborate on issues related to applying the thick control flow concept on OO constructions and concepts. We see straightforward possibilities; e.g. as the calling object can be seen as the 0th parameter, we can easily support thick object values as callers.

3. An example program

To get an idea, how the thick control flow concept could be embedded to an imperative language, we discuss of an example written on a hypothetical extension of Scala.

Matrix multiplication of $A_{h \times n} \times B_{n \times w}$ yields a matrix $C_{h \times w}$, where

$$c_{i,j} = \sum_{k=1}^n a_{i,k} \times b_{k,j}.$$

An obvious solution would be to compute each $c_{i,j}$ in parallel with a sequential algorithm following the definition. This would yield an algorithm running in $O(n)$ time with $h \times w$ implicit threads working with the CREW PRAM memory model. By carefully timing the references to matrix elements, the obvious algorithm can be made to work on an EREW PRAM, too.

The example below defines two functions related to multiplication: `matrix_mul` and `matrix_mul_maxthick`. The latter simply defines a thick block of thickness $h \times w$ and makes a thick function call. The called function does not use replicated formal parameters, but rather the call thickness is indirectly seen to relate to input and output values. The thickness `#` is assumed to `#|(h × w)`, and the idea is to calculate by one implicit thread an area of $h \times w / \#$ values of the return matrix C .

Observe that the first 4 assignment statements of `matrix_mul` are non-replicated where as the fifth is replicated! Both of the while-loops are executed synchronously with the thickness `#` – as the loop-condition is not a replicated expression, no flow branching will take place. Notice also that in the inner while-loop, the first assignment statement is replicated whereas the second is not.

```
// @ = flow idx inside control flow
// # = control flow thickness
```

```
// h x w must be a multiple of #
def rep matrix_mul[T: Numeric]
  (in A: Array[Array[T]],
   in B: Array[Array[T]],
   out C: Array[Array[T]]) {
  val h = A.length
  val n = A[0].length
  val w = B[0].length

  val block_size = h*w / #
  rep val block_idx = @ * block_size

  var i = 0
  while (i < block_size) {
    rep val idx = block_idx + i
    i += 1

    rep val x = idx % w
    rep val y = idx / w

    var j = 0
    while (j < w) {
      C[y][x] = C[y][x] + A[y][j]*B[j][x]
      j += 1
    }
  }
}

def matrix_mul_maxthick[T: Numeric]
  (in A: Array[Array[T]],
   in B: Array[Array[T]],
   out C: Array[Array[T]]) {
  val h = A.length
  val w = B[0].length

  replicate(h*w) {
    matrix_mul(A, B, C)
  }
}

// example
def main(w: Int, n: Int, h: Int) = {

  def gen_matrix(w: Int, h: Int) =
    Array.fill(h)(Array.fill(w)(0))

  val A = gen_matrix(n, h)
```

```

val B = gen_matrix(w, n)
val C = gen_matrix(w, h)

matrix_mul_maxthick(A, B, C)
}

```

4. Implementation considerations and related work

Implementation of a thick control flow at the executing multicore processor may happen as is usual today: Each thread of a control flow of thickness t is represented as a single thread. A clear challenge is how to efficiently implement the implicitly synchronous execution. Without going into details, efficient implementations are possible by using so-called synchronization wave [9], [8] and parallel slackness [10]. Such PRAM realizations studies are done e.g. in [2], [1], [8], [6], [5]. However, such an implementation would not advance the full potential of the thick control flow approach (not all expressions are thick) and handling a huge number of implicit threads would be very challenging.

In XMT / ParaLeap / ICE [12], [11], Vishkin has cleverly solved the issue of supporting a huge number of virtual threads. For a parallel block, the solution extracts a set of threads from the pool of unexecuted threads (concerning the block), spreads them for the thread execution units (the used terminology is different), makes them to efficiently execute those threads to the end, and then extracts a new set of threads. If the parallel block has a lot of statements, then there is a risk that the statement-wisely non-synchronous execution is not able follow a PRAM-style memory consistency.

The idea of XMT / ParaLeap / ICE could be applied to thick control flows but still maintaining the statement-wise synchrony. The hardware would need to support a small number of parallel control flow executions, each of arbitrary thickness. Assume that there would be z SIMD-style execution units of some fixed width w . Now, the hardware should just try to extract, from the arbitrarily thick control flows, z thick instructions of width at most w , feed those to instruction execution units and repeat. Although the synchronous execution with instruction could be guaranteed this way, the joining of flows requires a specific solution (synchronization wave). In principle, in multicore systems the ordinary function of cores as thread execution units would need to be replaced with

a rather similar function of thick control flow execution units.

Naturally, the efficiency of thick control flow based execution is heavily dependent on the ability to advance memory hierarchy efficiently and hide memory access latencies. Such solutions in connection of stream computing are discussed e.g. in [7]. The whole idea of thick control flows is very close to that of vector and stream computing and e.g. the Brook language [3], [7]. Many of the GPGPU computing related approaches (BrookGPU, OpenCL, CUDA) can be seen to have a lot in common with out thick control flows. E.g. the stream based BrookGPU [3] in practice defines computational functions (called kernels) that operate on multiple streams and produce stream values. The streams can have a multidimensional shape and that shape corresponds to a set of executing threads. Executing a kernel means synchronously executing a thick control flow (the kernel's body) over the stream values. At execution level, the SIMT (Single Instruction Multiple Thread) approach of GPU devices is of course close to our approach. The main difference with respect to stream computing is the dataflow/functional style versus the imperative style of thick control flows.

We consider there to exist lots of possibilities for implementing the thick control flow approach efficiently, yet we also agree that some level of guidance (annotations, etc) might be required from the programmer. However, at the moment we must leave the architectural issues open as this is work-in-progress.

5. Conclusions

We have presented the idea of thick control flows. Besides program correctness, the technical software problem of many contemporary approaches to multithreaded computations is how to organize the co-operation of a huge number of parallel threads. For these software problems, we consider parallel thick control flows as a good solution. There are reasons to believe that efficient execution of thick control flows would be possible, but in this work-in-progress paper, we need to leave that issue open.

References

- [1] F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer. On the Physical Design of PRAMs. *The Computer Journal*, 36(8):756 – 762, 1993.

- [2] F. Abolhassan, J. Keller, and W.J. Paul. On the Cost-Effectiveness of PRAMs. In *Proceedings, 3rd IEEE Symposium on Parallel and Distributed Computing, ACM Special Interest Group on Computer Architecture, and IEEE Computer Society*, pages 2 – 9, 1991.
- [3] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *Proceedings of SIGGRAPH*, 2004.
- [4] R.D. Blumofe et al. Cilk: An Efficient Multithreaded Runtime System, 1995.
- [5] M. Forsell. *TOTAL ECLIPSE – An Efficient Architectural Realization of the Parallel Random Access Machine* booktitle=.
- [6] M. Forsell. A scalable high-performance computing solution for network-on-chips. *Micro, IEEE*, 22(5):46 – 55, sep–oct 2002.
- [7] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W.J. Dally. Architectural Support for the Stream Execution Model on General-Purpose Processors.
- [8] V. Leppänen. *Studies on the Realization of PRAM*. PhD thesis, University of Turku, TUCS, Lemminkaisenkatu 14, FIN-20520 Turku, Finland, nov 1996. TUCS Dissertations No 3.
- [9] A.G. Ranade. How to Emulate Shared Memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.
- [10] L.G. Valiant. General Purpose Parallel Architectures. In *Algorithms and Complexity, Handbook of Theoretical Computer Science*, volume A, pages 943–971, 1990.
- [11] U. Vishkin. Using Simple Abstraction to Reinvent Computing for Parallelism. *Communications of the ACM*, 54(1):75–85, 2011.
- [12] Uzi Vishkin, Shlomit Dascal, Efraim Berkovich, and Joseph Nuzman. Explicit multi-threaded (xmt) bridging models for instruction parallelism. In *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 140–151, 1998.

Dynamic Workflow Composition and Execution

Binh Minh Nguyen, Viet Tran, Ladislav Hluchy
Institute of Informatics, SAS, Slovakia

Abstract- In this paper, we present a new approach for programmable workflow composition. The workflow is represented as sequence of tasks with explicitly defined input/output data. Parallelism between tasks is implicitly defined by the data dependence. Users can create a more complicated workflow by scripts, including nested and parameterized workflows. The workflow will be executed in distributed workflow managers.

I. INTRODUCTION

With the advance of computational technologies, the scientific applications running on modern distributed systems became more and more complex. Each execution of the applications is usually a workflow of several connected steps, where the output of the previous steps are the input of the next steps. Therefore, the tasks have to be executed in the correct order and the data need to be transferred between tasks in order to get the correct results.

At the moment, there are many existing workflow management systems, each system has its own language for describing the workflows. The way how the workflows are described in current systems are rather complex and inflexible. Some systems come also with graphical editors for creating the workflows easier.

In this paper, we present a new approach for creating and executing workflows for scientific applications. Also our approach is applicable elsewhere, we primarily focus on distributed systems, where each task is an execution of a program (script, binary executable) on target hardware platforms. Most of grid workflow management systems have the same characteristics, so we will compare our approach with these workflow managers.

II. OVERVIEW OF WORKFLOW DESCRIPTION APPROACHES

Each workflow description consists from two parts: description of tasks and description of dependences between tasks. Each task may have several properties like execution code, input/output data, command-line arguments, requirements on hardware and so on. There are two main approaches to describe these properties of tasks: in a plain text form as pairs of property name and value (e.g. *CPU*Number = 4), or in XML language where task properties are elements or attributes.

Beside the task description, the dependence between tasks in the workflows must be also described in the workflow languages. There are two main ways to describe dependence between tasks in workflows: using parallel/sequence instructions and using directed acyclic graphs.

In the first approach, a workflow is consisted of (nested) parallel or sequential blocks of tasks. Tasks that can be executed in parallel are placed in blocks with parallel instruction, otherwise, in a block with sequential instruction, the tasks must be executed in the order as they are defined in the block. An example of workflow described in this way is as follows:

```
SEQ
  Task1
  PAR
    Task2
    Task3
  Task4
```

In this example, the workflow has four tasks named *Task1*, ..., *Task4*. The first task *Task1* must finish before *Task2* and *Task3* can start. *Task2* and *Task3* can be executed in parallel (or in any order), and *Task4* must wait until both tasks finish. For example Karajan [3] in Cog Kit [4] uses this approach for describing workflows.

In the second approach, the dependences between tasks are described as by parent-child pair. Children tasks must wait until all parent tasks finish before starting. The workflow above can be described in this approach as follows:

```
PARENT Task1 CHILD Task2, Task3
PARENT Task2, Task3 CHILD Task4
```

Majority of scientific workflows use this approach for describing dependence. Typical examples are JDL (Job Description Language) [1] which are used by gLite [2], DAGMan [5] in Condor [6], SCULF [7] in Taverna [8], Pegasus [9]. The main advantage of this approach is that it can describe more complex workflows than the first approach. The dependences can be visualized as directed acyclic graphs (DAG), where tasks are represented by nodes of the graphs and the directed edges show the parent-child relationships. Fig. 1 shows the graph of the workflow in the example above.

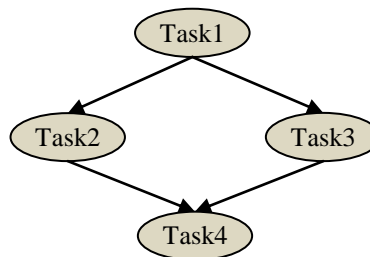


Fig.1 Workflow represented as DAG

Except for the mentioned two approaches, there are also other ways for describing workflows, e.g. GWorkflowDL [10] in GWES uses Petri net for workflow representation, however, these approaches are less common.

Another thing about workflow representation is workflow editor. Many workflow engines use XML-based languages for describing workflows. Although XML is excellent for machine processing, they are not suitable for human reading. Editing a large workflow in XML by using text editors is very challenging. Therefore, several projects e.g. Taverna, GWES, offer graphical workflow editor for creating/editing workflows. Although these graphical editors can help users to visualize workflows and edit them, they have one major drawback: they are difficult to make scripts, e.g. parameterized workflows or repeated tasks.

III. PROGRAMABLE WORKFLOW DESCRIPTION

We use a simpler way to describe workflows as follows:

- A task in a workflow is described by triple: its code, a set of input data and a set of output data.
- A workflow is described as a sequence of tasks.
- Dependence and parallelism among tasks are implicitly defined by the input/output of tasks.

An example of a workflow is follows:

```
Myworkflow(input, output, N)
  Task(preprocess, input, local)
  for i = 1 to N
    Task(simulation, local, result[i])
  Task(postprocess, result, output)
```

In the code above, the parameters *input* and *output* of the workflows are the lists of input and output data of the workflow and *N* is an additional literal parameter of the mentioned workflow. As it is shown, the workflow is parameterized: users can define input/output data of the workflows at the runtime, and some other literal parameters if exist. The items in the lists of input/output data are usually the names of files containing corresponding data. The differences between input data and additional parameters of workflows are that: the values of additional parameters must be known at the moment the workflows is created and when the real values of input/output data (i.e. the contents of files in the lists) will be known at the moment the workflow is executed.

The first task uses code in file *preprocess* for processing data from *input* and produces data stored in files in *local*. The loop will create *N* tasks which run code in *simulation* with *local* as input data and create *N* results indexed as *result[1]*, ..., *result[N]*. Finally the last task use data from the list *result*, and create output of whole workflow.

As it is shown in the example above, we only describe tasks, not the dependences among tasks. The dependence is implicitly defined by the input/output data of tasks. For example, second task use data produced by first task, so it must wait until the first task finishes.

We use Python scripting language for implementing a workflow composition tool for processing workflow description in our approach. The use of Python is inspired by Ganga job management [11] developed by CERN. In fact, each workflow, described in examples above, is a function in Python. With Python, we can easily write define workflows with loops and/other control instructions.

Internally, the workflow internally consists of two lists: list of tasks and list of data. Each task in the workflow is an object in memory with references to its data. Fig. 2 shows the internal memory structures of workflows: the list of tasks on the left side, the list of data on the right side, and references between tasks and data.

It is worth to note that the graph in Fig. 2 can be generated with linear complexity. We don't have to analyze every pair of tasks to know if they have data dependence, but just read the task description and make connection to the its data. As every task is read only once, the complete graph can be generated with linear complexity.

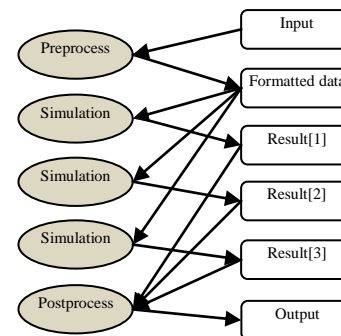


Fig. 2 Internal structure of workflow

It is interesting to see that the internal data structure of the workflow in Fig. 2 is exactly the graphical representation of the workflow like in Fig. 3. It means that, once the workflow description is processed, we have already graphical representation of the workflow in DAG form in memory. Therefore, it is easy to export the workflow description to any other formats compatible with DAG.

We can prove the equivalence of workflows in our approach and workflows described by directed acyclic graphs by following statements:

- Every workflow represented in DAG can be described in our approach.
- Every workflow described in our approach can be converted to DAG with linear complexity $O(N)$.

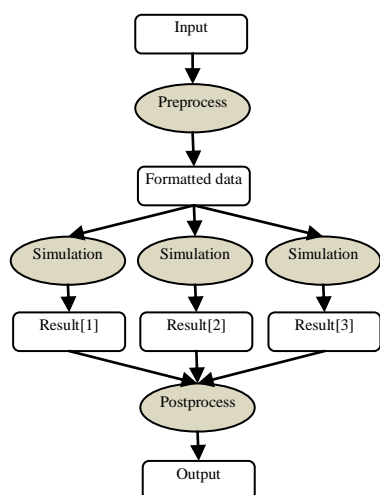


Fig. 3 Graphical presentation of workflow

The proof of the first statement is simple: just define the data transferred along a directed edge in the DAG as a data item (file) and we will have whole workflow as a list of tasks with input/output data for each task like in our approach. The proof of second statement can be derived from graphical representation of our approach.

By these statements we can prove the equivalence between our representation of workflow and DAG, they can be simply converted from one format to the other. The complexity of the conversion is $O(N)$, i.e. the cost of conversion per a task is constant and independent from the size of workflow. In our implementation, the cost is only some microseconds. In comparison with typical execution times of tasks in grid computing (several minutes) we can safely say that the cost is really negligible.

IV. WORKFLOW SCHEDULING AND EXECUTION

A processing element is a hardware entity where tasks are executed. It can be a CPU, computer node or cluster in grid computing or a server where services are located.

A group of processing elements (PE group) is a set of PEs where the times needed for data transfers between tasks in a workflow is negligible in comparison with the execution times of tasks.

We are saying about distributed computing when the times needed for data transfers between tasks located in different PE groups in a workflow are significant in comparison with the execution times of tasks. If tasks in the workflow are not well scheduled, it can cause serious performance penalties because of data transfer. Therefore, optimization of workflow execution is very important.

There are many scheduling algorithms that can optimize workflows execution. In [12], a nice taxonomy with scheduling strategies was presented. It is proved in [13] that a good scheduling algorithm can significantly improve performance of workflows execution.

One of the main weaknesses in design of existing scientific workflow management systems is that there is only single central workflow manager that handles all communications,

dispatches tasks and monitors status of their executions. Even successive tasks are mapped on the same processing element (PE), tasks are usually send output data to the workflow manager and then the data are sent back to the PE for following tasks. Such communication is very inefficient and in distributed environment with high latency, that can cause serious performance penalties.

V. DISTRIBUTED WORKFLOW EXECUTION

In our approach, each group of processing elements has a local workflow manager which takes care of task execution on the PE group and handles all data used by the tasks. Each local manager has complete workflow with schedule. Communication can be realized between local managers asynchronously.

The central workflow manager is responsible for global scheduling and starting the local managers. We adopt the scheduling approach described in [14] for scheduling tasks in the workflows. Once the schedule is done and the local workflow managers running on PE have started, the central manager sends the whole workflow with schedule to every local manager.

The local managers will receive the workflows with scheduling and execute tasks that have been assigned to their own PE. Data transferred between tasks will be realized directly between local managers. As data used and produced by a task is explicitly declared in the workflow description, the local workflow manager can know where to send data.

We choose Java programming language for implementation of central and local workflow managers due to its excellent network libraries. The local managers run as a background process in a PE. Each local manager has two main modules: communication module for handling communication with central and other local managers, send and receive data, and execution module: handling task execution.

Local manager use very little CPU, most of time they are in waiting status (communication module waiting for data, execution module wait for task finish), so they do not need separate CPU for running, but share CPU for task execution.

The central manager steers the execution of workflows at local managers via these commands:

- **CANCEL**: cancel workflow execution and terminate local managers
- **RECONFIG**: the central manager send new schedule (for unexecuted tasks) to local managers. It can happen when some local managers do not respond due to fault occurrence on PE, so tasks assigned to the PE need to redistribute.
- **STATUS**: getting status of tasks on the local managers. The central manager also uses STATUS command as heartbeat signal to detect fault occurrence at the local managers.
- **GETDATA**: getting data from local managers. By default, the local managers do not send intermediate data to the central managers.

V. CONCLUSION AND FUTURE WORK

In this paper, we have presented our approach for flexible workflow composition. Workflows are described as a list of tasks with input/output data explicitly defined. This approach supports nested workflows, loops and other control instructions in Python scripting language, which is used in our implementation. The workflows then can be executed by distributed workflow management system.

ACKNOWLEDGMENT

This work is supported by projects SMART ITMS: 26240120005, SMART II ITMS: 26240120029, VEGA 2/0184/10.

REFERENCES

- [1] E. Laure et al. Programming the Grid with gLite. Computational methods in science and technology. Vol. 12, No. 1, pp. 33-45, 2006.
- [2] gLite - Lightweight Middleware for Grid Computing. <http://glite.cern.ch>. 2011.
- [3] Gregor von Laszewski, Mihael Hategan and Deepti Kodeboyina. Java CoG Kit Workflow. Workflows for E-Science, Part III, pp. 340-356, 2007.
- [4] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java Commodity Grid Kit. Concurrency and Computation: Practice and Experience, 13(89), pp. 643- 662, 2001.
- [5] J.Frey. Condor DAGMan: Handling inter-job dependencies. 2002.
- [6] Condor project homepage. <http://www.cs.wisc.edu/condor/>. 2010.
- [7] Kostas Votis et al. Workflow coordination in grid networks for Supporting enterprise-wide business Solutions. IADIS International Conference e-Commerce, pp. 253-260, 2004.
- [8] 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, iss. Web Server issue, pp. 729-732, 2006.
- [9] K.Lee, N. W. Paton, R. Sakellariou, E. Deelman, A. A. A. Fernandes, G. Mehta. Adaptive Workflow Processing and Execution in Pegasus. 3rd International Workshop on Workflow Management and Applications in Grid Environments pp. 99-106, 2008.
- [10] Martin Alt, Andreas Hoheisel, Hans-Werner Pohl and Sergei Gorchatch. A Grid Workflow Language Using High-Level Petri Nets. Parallel Processing and Applied Mathematics, pp.715-722, 2006.
- [11] K. Harrison, C.L. Tan, D. Liko, A. Maier, J. Mościcki, U. Egede, R.W.L. Jones, A. Soroko, G.N. Patrick. Ganga: a Grid User Interface. International Conference on Computing in High Energy and Nuclear Physics, pp. 982-985, 2006.
- [12] Jia Yu and Rajkumar Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. SIGMOD Record, Vol. 34, No. 3, pp. 44-49, 2005.
- [13] Marek Wiecezorek, Radu Prodan and Thomas Fahringer. Scheduling of Scientific Workflows in the ASKALON Grid Environment. SIGMOD Record, Vol. 34, No. 3, pp. 56-62, 2005.
- [14] Yili Gong, Marlon E. Pierce, and Geoffrey C. Fox. Dynamic Resource-Critical Workflow Scheduling in Heterogeneous Environments. Workshops on Job Scheduling Strategies for Parallel Processing, pp. 1-15, 2009.

Predicting CPU Availability of a Multi-core Processor Executing Concurrent Java Threads

Khondker Shajadul Hasan¹, Nicolas G. Grounds², John K. Antonio¹

¹School of Computer Science, University of Oklahoma, 110 W. Boyd St., Norman, OK 73019, USA
shajadul@ou.edu, antonio@ou.edu

²MSCI, 201 David L Boren Blvd # 300, Norman, OK 73072, USA
nicolas.grounds@msci.com

Abstract - Techniques for predicting the availability of CPU resources associated with the execution of multiple concurrent Java threads on a multi-core architecture are introduced. Prediction of CPU availability is important in the context of making thread assignment and scheduling decisions. Theoretically derived upper and lower bound formulas for estimating CPU availability are introduced. Input parameters to the formulas include: number of cores; number of threads; and the unloaded CPU usage factor for each thread. Extensive experimental studies and statistical analysis are performed to validate the theoretical bounds and provide a basis for an empirical model for predicting CPU availability. To facilitate scientific and controlled empirical evaluation, synthetically generated threads are employed that are parameterized by their unloaded CPU usage factor, defined as the fraction of time a thread spends utilizing CPU resources on an unloaded system.

Keywords: Concurrent threads; CPU availability; Multi-core processors; Java Virtual Machine.

1. Introduction

Multithreading is a common technique used for exploiting performance from multi-core processors. When the number of threads assigned to a multi-core processor is less than or equal to the number of CPU cores associated with the processor, then the performance of the CPU is predictable, and is often nearly ideal. When the number of assigned threads is more than the number of CPU cores, the resulting CPU performance can be more difficult to predict. For example, assigning two CPU-bound threads to a single core results in CPU availability of about 50%, meaning that roughly 50% of the CPU resource is available for executing either thread. Alternatively, if two I/O-bound threads are assigned to a single core, it is possible that the resulting CPU availability is nearly 100%, provided that the usage of the CPU resource by each thread is fortuitously interleaved. However, if the points in time where both I/O-bound threads do require the CPU

resource overlap (i.e., they are not interleaved), then it is possible (although perhaps not likely) that the CPU availability of the two I/O bound threads could be as low as 50%. Predicting the availability of CPU resources when the number of threads assigned to the processor exceeds the number of cores is important in making thread assignment and scheduling decisions. Precise values of CPU availability are difficult to predict because of a dependence on many factors, including context switching overhead, CPU usage requirements of the threads, the degree of interleaving of the timing of the CPU requirements of the threads, and the characteristics of the thread scheduler of the underlying operating system (or Java Virtual Machine). Due to the complex nature of the execution environment, an empirical approach is employed to evaluate proposed CPU availability prediction models and formulas.

The success of approaches for assigning threads (or processes) to multi-core systems relies on the existence of reasonably accurate models for estimating CPU availability. This is because there is a strong relationship between a thread's total execution time and the availability of CPU resources used for its execution. Therefore, predicting the CPU availability that results when threads are assigned to a processor is a basic problem that arises in many important contexts [1].

In the present paper, an extensive collection of empirical measurements taken from both single- and multi-core processors provide a basis for validating proposed analytical models for estimating CPU availability. The proposed models are theoretically derived upper and lower bound formulas for CPU availability. Confidence interval and moving average statistics from measured CPU availability (from different empirical case studies) validate the utility of these theoretical models. Case studies for a single core machine involve spawning 2, 3 and 4 concurrent threads with randomly selected CPU usage factors. Each case study includes a collection of about 2,000 sample executions. For a quad core machine, similar

case studies are conducted involving 8, 12 and 16 concurrent threads, again with randomly selected CPU usage factors for the threads.

An interesting outcome of the empirical case studies is that the variation in measured CPU availability is very low whenever the sum of the threads' CPU usage factors is either relatively low, or relatively high. Thus, prediction of CPU availability is quite accurate when the CPU is either lightly or heavily loaded. When the total CPU loading (i.e., sum of all threads' CPU usage factors) is moderate, the realized availability of CPU resources has more variation and thus is less predictable. However, even the largest variation in CPU availability measured was typically no more than 20%, with a 90% confidence. The empirical studies also show that, as the number of concurrent threads increases, the context switching overhead degrades the performance of thread execution; resulting in a slightly wider gap between the theoretically derived upper bound and measured availability values. In both single- and multi-core cases, when the number of threads is above the number of CPU cores, the performance of thread execution is predicted reasonably well by the theoretical upper bound formula for CPU availability.

The rest of the paper is organized in the following manner. Section 2 discusses relevant background related to the execution of Java threads, and motivates the importance of predicting CPU availability. Section 3 introduces the specific thread execution model assumed in this paper; from this model, theoretical derivations of upper and lower bound formulas for CPU availability are provided. Section 4 presents the empirical studies including benchmarking, case study measurements for single- and multi-core CPUs, and statistical analysis of the results. Finally, Section 5 contains concluding remarks and suggestions for future work.

2. Execution of Concurrent Java Threads

The primary focus of this paper is to estimate the CPU resource availability of a Java Virtual Machine (JVM) executing concurrent threads. The concept of a Java monitor [2] is useful for describing how threads are executed by a JVM. A graphical depiction of such a monitor is shown in Figure 1, which contains three major sections. In the center, the large circular area represents the owner, which contains two active threads, represented by the small shaded circles. The number of threads the owner can contain (concurrently) is bounded by the number of CPU cores. At the upper left, the rectangular area represents the entry set. At the lower right, the other rectangular area represents the wait set. Waiting or suspended threads are illustrated as striped circles.

Figure 1 also shows several labeled "doors" that threads must pass through to interact with the monitor. In the figure, one thread is suspended in the entry set and one thread is suspended in the wait set. These threads will remain where they are until one of the active threads releases its position in the monitor. An active thread can release the monitor in either of two ways: it can complete the monitor region it is executing or it can execute a wait command. If it completes the monitor region, it exits the monitor via the door labeled E. If it executes a wait command, it releases the monitor and passes through door labeled C, the door into the wait set.

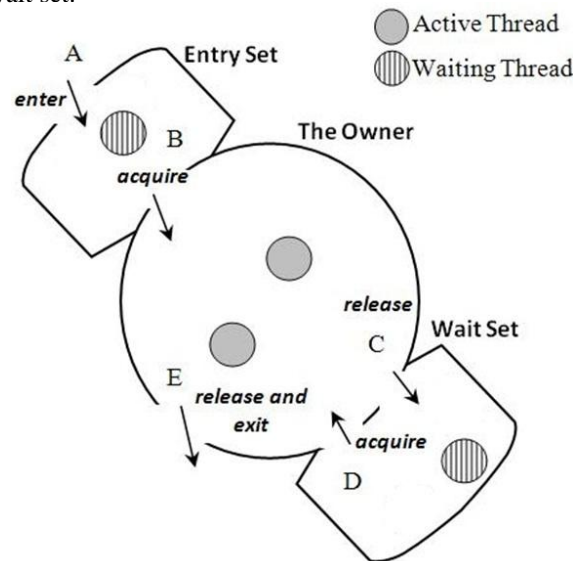


Figure 1: A typical Java Monitor, derived from [2].

To illustrate issues associated with predicting CPU availability, assume the monitor of Figure 1 is associated with a dual-core JVM. From the figure, there are a total of four threads assigned to the JVM; two are currently in a waiting state and two are currently active. If all four threads are CPU-bound, meaning that their unloaded CPU usage factor is 100%, then it is clear that the availability of the CPU resources (i.e., both CPU cores) will be about 50%. If, on the other hand, some threads are I/O-bound (e.g., having CPU usage factors less than say 25%) then predicting CPU availability is not as straightforward. In general, the realized CPU availability depends upon the scheduling scheme employed by the JVM (and/or the underlying OS) for transitioning threads between the active and waiting states.

In the next section, an analytical framework is developed for estimating CPU availability associated with executing concurrent threads on a multi-core JVM. The primary contribution of the section is the derivation of upper and lower bound formulas for CPU availability.

3. Bounds for Multi-core CPU Availability

In this section, an analytical framework is developed for estimating CPU availability associated with executing concurrent threads on a multi-core processor. The primary contribution of this section is the derivation of upper and lower bound formulas for CPU availability. A thread is modeled by a series of alternating work and sleep phases. For the purposes of this study, the work portion of a phase is CPU-bound and requires a fixed amount of computational work (i.e., CPU cycles). The sleep portion of a phase does not consume CPU cycles, and its length relative to that of the work portion is used to define the CPU load usage factor for a thread. Figure 2 shows three work-sleep phases of a thread.

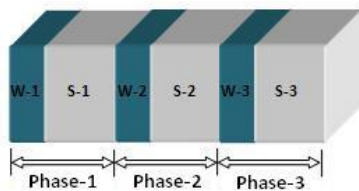


Figure 2: Three work-sleep phases of a thread.

In the assumed framework, a thread in the work portion of a phase will remain in the work portion until it has consumed enough CPU cycles to complete the allotted work of that portion. After completing the work portion of a phase, the thread then enters in the sleep portion where it sleeps (does not consume CPU cycles) for an amount of time defined by the CPU usage factor. When multiple threads are spawned concurrently, the JVM runs those threads employing a time sharing technique (refer to the discussion of the Java Monitor in Section 2). The CPU availability (and performance) will be degraded when the work phase of all threads overlap each other in time. Figure 3 depicts a scenario where 3 threads with identical work-sleep phases are executed in a single-core execution environment.

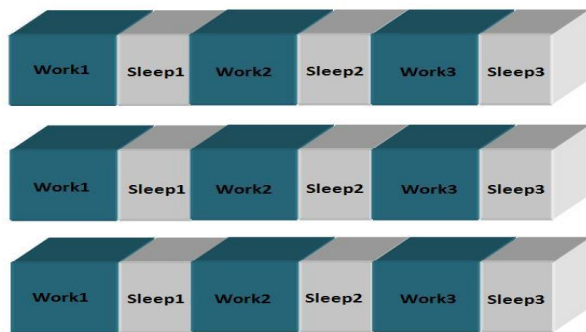


Figure 3: Three concurrent threads having identical work-sleep phases and CPU usage factors executing on a single-core environment.

Each thread gets a maximum of 1/3 of the available CPU resource during the work portions of their phases, resulting in 3-times wider work portion execution time than would be the case for a single thread scenario. Alternatively, if the work portions of these three threads are staggered to where there is no overlap, then there is no contention for the CPU resource and the CPU availability is essentially 100%, as shown in Figure 4. That is, all the work phases of concurrent threads are separated in time so that each thread can get the full usage of the CPU the moment it is first needed by each thread. The ideal phasing illustrated in Figure 4 requires that the work of any two threads can be accomplished within the time of one sleep portion of a phase.

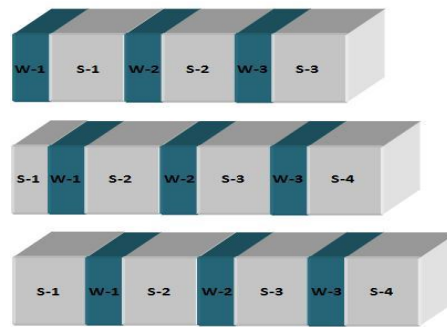


Figure 4: Three concurrent threads with identical work-sleep phases and CPU usage factors with non-overlapping work portions.

In the proposed framework, each thread is parameterized by a CPU usage factor and a total amount of computational work to accomplish. The CPU usage factor is defined as the time required to complete the work portion of a work-sleep phase on an unloaded CPU, divided by the total time of a work-sleep phase. A thread having zero sleep length has a CPU usage factor of 100%, which is also called a CPU-bound thread.

For a single core machine, the following formulas define upper and lower bounds for CPU availability. Here, n is the number of threads assigned to the single core machine and L is the aggregate loading factor, defined as the sum of CPU usage factors of all n threads. The upper bound formula for CPU availability is:

$$\bar{a} = \frac{1}{\text{Max}(1, L)} = \begin{cases} 1, & \text{if } L < 1 \\ 1/L & \text{if } L \geq 1 \end{cases} \quad (1)$$

and the lower bound formula is:

$$\underline{a} = \frac{1}{1 + \left(\frac{n-1}{n}\right) \times L} \quad (2)$$

The upper bound formula represents the best case CPU availability, which is illustrated by the example of Figure 4 in which none of the threads use the CPU

resource concurrently. Provided that the sum of the usage factors of the threads is less than unity, then it is possible that the CPU availability could be as high as unity (i.e., 100%). When the sum of the CPU usage factors is greater than unity, then the best possible value for CPU availability is $1/L$. The lower bound formula is associated with a situation in which the threads' usage of the CPU resource has maximum overlap, as depicted in Figure 3 in which all threads always use the CPU resource concurrently.

For a multi-core machine with c cores, the following formulas define upper and lower bounds for CPU availability. The upper bound formula is:

$$\bar{a} = \frac{1}{\text{Max}(1, L/c)} = \begin{cases} 1, & \text{if } L/c < 1 \\ c/L & \text{if } L/c \geq 1 \end{cases} \quad (3)$$

and the lower bound is:

$$\underline{a} = \frac{1}{1 + \binom{n-1}{n} \times \left(\frac{L}{c}\right)} \quad (4)$$

Note that Eqs. 3 and 4 are generalizations of Eqs. 1 and 2, i.e., for the case $c = 1$, Eqs. 3 and 4 are identical to Eqs. 1 and 2.

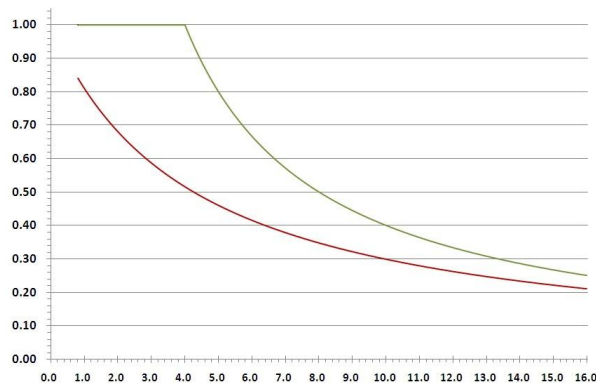


Figure 5: Upper and lower bounds for CPU availability prediction for $c = 4$ cores.

Figure 5 shows plots of the upper and lower bound formulas for the case $c = 4$. Note that the difference in upper and lower bounds can be significant for moderate values of aggregate loading. In the following section, experimental studies are performed to determine actual measured values of CPU availability in relation to these bounds.

4. Experimental Studies

4.1 Overview

The purpose of the experimental studies is to empirically measure CPU availability as a function of aggregate loading for collections of threads with

randomly selected CPU usage factors. For the study, threads are generated synthetically so that their CPU usage factors can be set accurately. The measured CPU availability associated with a collection of threads executing on a processor is defined by the ratio between the ideal time required to execute one of the threads on an unloaded processor divided by that thread's execution time on a loaded processor. About 2,000 randomly selected collections of threads are generated for each study; each randomly generated collection of threads provides one measurement of CPU availability. The major part of the experimental system's flow control is shown in Figure 6.

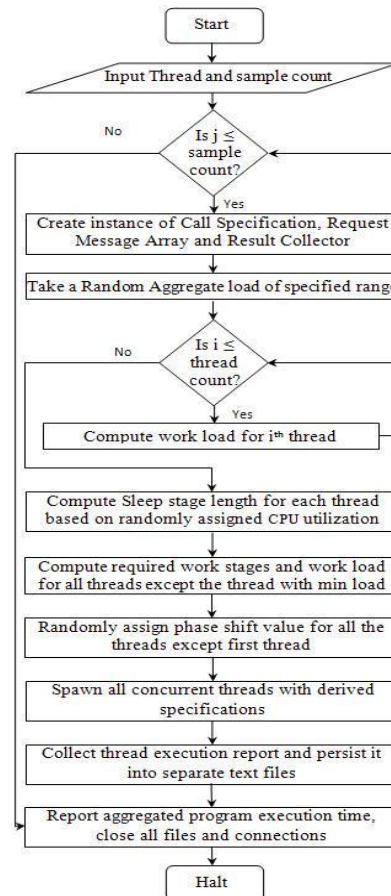


Figure 6: Flow chart for load distribution and execution process of threads.

To ensure a uniform sampling of data across the values of possible aggregate loadings, a random value of aggregate loading is chosen first. For example, for a scenario having two threads, a value of aggregate loading is chosen between a small value ϵ and 2; denote this value as L . Then a random value is chosen between $\max\{\epsilon, L-1\}$ and $\min\{1, L\}$, which defines the CPU usage factor of the first thread, say T_1 ; the CPU usage factor of the second thread is then

defined as $T_2 = L - T_1$. In general, for n threads, the following algorithm is used to randomly assign the CPU usage factors for a given value of aggregate loading L :

```

For  $i = 1$  to  $n - 1$  do
   $LB = \text{Max}((L - \sum_{j=1}^{i-1} T_j - (n - i)), \epsilon)$ 
   $UB = \text{Min}((L - \sum_{j=1}^{i-1} T_j - (n - i) * \epsilon), 1)$ 
  Pick  $T_i$  from the following range
   $T_i \in [LB, UB]$ 
   $\Delta = UB - LB$ 
   $T_i = \Delta \times T_i + LB$ 
End
 $T_n \leftarrow L - \sum_{i=1}^{n-1} T_i$ 

```

Recall that the CPU usage factor of a thread defines the relative amount of work performed during its work phase relative to its sleep phase. As described in the flow chart of Figure 6, a random phase shift is also assigned to each thread. The range of the phase shift value is from 0 to length of the phase. This random phase shift provides a degree of staggering of phases among the threads. Figure 4 is an example of the best case staggering; whereas Figure 3 is an example of worst case staggering in terms of CPU availability.

4.2 Experimental Environment

The system used for evaluating the single core test cases is an Intel Xeon CPU E5540 @ 2.53GHz clock speed, 1,333 MHz bus speed and 4 GB of RAM. The system used for evaluating the multi-core test cases is an Intel Core 2 Quad (quad-core processor), 2.83GHz clock speed, 1,333MHz bus speed with 4 GB of RAM. Due to the different configurations of the single- and quad-core systems, benchmarking for the single-core and quad-core machines were determined separately. The JVM used for these experiments is JDK 1.6. Threads deployed here are independent tasks, meaning there are no interdependencies among threads such as message passing. Threads are spawned concurrently with workloads and phase shifts as described in the previous section. When a collection of threads completes, a report of the threads' execution is produced, which contains start time, work time, sleep time, number of phases, and end time.

4.3 Benchmarking

Benchmarking an unloaded system enables the calculation of parameters associated with the work and sleep portions of the phases to synthesize a particular CPU usage factor. For setting up the benchmarks, each thread was assigned 2.0×10^8 units of synthetic CPU work. Threads need to accomplish this total work in 50 work-sleep phases. In the work component of each phase, these threads accomplish 4.0×10^6 units of

work. For calculating the CPU usage factors (5%, 10%, 15%, etc.), the sleep phase length is varied using the following equation.

$$sleep = \left(\frac{stage\ work\ time \times (1 - CPU\ usage)}{CPU\ usage} \right) \quad (5)$$

According to the formula above, the sleep time increases as the CPU usage decreases. Thus, a thread with low CPU usage will sleep longer than other threads having a higher CPU usage factors.

4.4 Empirical CPU Availability Case Studies

For measuring the CPU availability of the single-core processor, three case studies were conducted in which multiple (2, 3, and 4) threads were spawned concurrently. An aggregate CPU load L was selected randomly, and distributed among the threads as described in Section 4.1.

Figures 7 and 8 show measured CPU availability scatter graphs for 2 and 4 concurrent threads executing on the single-core processor, superimposed with the plots of the upper and lower bound formulas derived in Section 3. In these figures, the horizontal axis represents aggregated CPU load and the vertical axis represents CPU availability. Each small dot in these graphs is an independent test case measurement of CPU availability. There are 2,000 dots in each figure representing measured CPU availability value among the concurrent threads. A moving average line is also drawn through the data on the graphs for helping to visualize the average measured performance. A window size of 0.10 aggregate CPU load and incremental value of 0.01 was used to calculate the moving average values. A similar sliding window approach was employed to calculate the 90% confidence interval upper and lower limits.

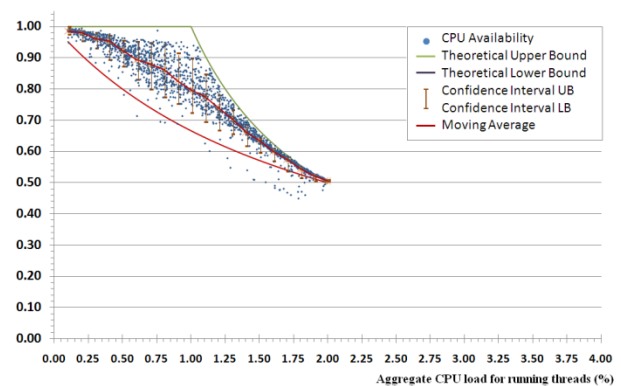


Figure 7: CPU availability of 2 threads in a single-core machine. Results of 2,000 independent test cases and 90% confidence intervals.

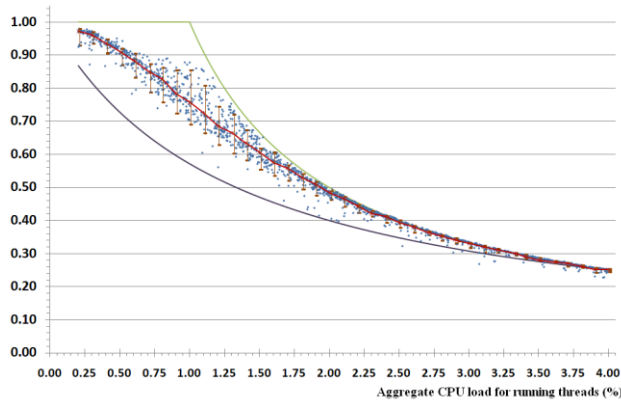


Figure 8: CPU availability of 4 threads in a single-core machine. Results of 2,000 independent test cases and 90% confidence intervals.

It is apparent from Figures 7 and 8 that the variation in CPU availability is low when the aggregate CPU loading is either relatively low or relatively high. Thus, CPU availability prediction is quite accurate when the CPU is either lightly or heavily loaded. When the total CPU loading is moderate, the measured CPU availability has more variation and thus it is less predictable. One of the intuitive reasons for low variation when threads have small CPU loading factors is their sleep phase lengths are wider, which decreases the probability of work portion overlap among the threads. On the other hand, when the aggregate CPU load is large, threads have smaller sleep phase and longer work phase lengths which almost always forces work phase to overlap and decrease performance, but in a predictable way.

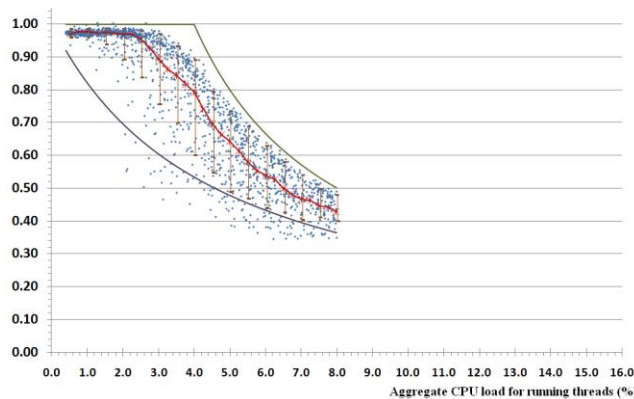


Figure 9: CPU availability of 8 threads in a quad-core machine. Results for 2,000 independent test cases and 90% confidence intervals.

For measuring the CPU availability of the quad-core processor, three case studies were conducted in which multiple (8, 12, and 16) threads were spawned concurrently to relatively compare the performance between single- and quad-core processors. A similar

approach has been employed to calculate the moving average and 90% confidence interval values. CPU availability graphs for 8 and 16 threads are shown in Figure 9 and Figure 10 respectively. Figure 9 shows a higher variation of CPU availability compared with Figure 7, which shows CPU availability of 2 threads in a single-core machine. During the thread execution life cycle, depending on CPU availability, threads might be allocated in different cores for load balancing which decrease the probability of work phase overlap.

The empirical results for the quad-core processor also show that the CPU availability prediction is quite accurate when the CPU is either lightly or heavily loaded. When the total CPU loading is moderate, the measured CPU availability has more variation and thus it is less predictable.

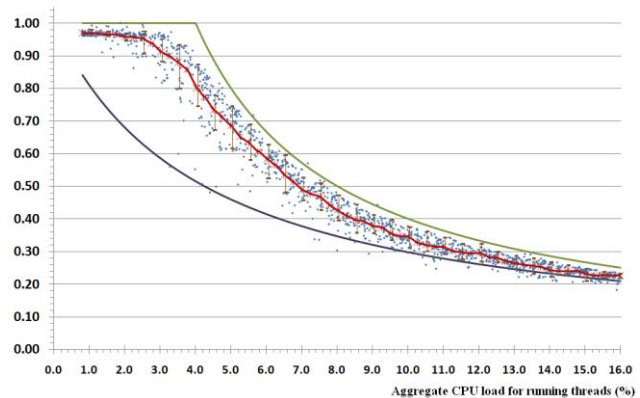


Figure 10: CPU availability of 16 threads in a quad-core machine. Results of 2,000 independent test cases and 90% confidence intervals.

The empirical results of the quad-core processor also shows that when the numbers of concurrent threads are exact multiples of the number of cores, it provides much better performance than when they are not. That measured variation can be as large as 10% of the overall CPU availability. From the empirical results of single- and multi-core processors, and measured CPU availability scatter graphs, it is apparent that theoretically derived upper and lower limits introduced in this paper bound actual measured values of CPU availability reasonably well.

In further reporting the results of the studies, it is convenient to define the normalized aggregate load, L/n , which is the aggregate load L normalized by the number of threads n . For sample values of normalized aggregate load, Table 1 shows the average measured CPU availability (Avg.), the difference in the upper and lower bound formulas (Bnd Diff) and the difference in the 90% confidence interval limits (CI Diff) for 2, 3 and 4 concurrent threads on the single-core processor.

Table 1: Average measured CPU availability (Avg.) for threads in a single-core machine along with differences of bound formulas (Bnd Diff) and 90% confidence interval limits (CI Diff).

L/n	2 Thread CPU Availability			3 Thread CPU Avail.			4 Thread CPU Availability		
	Avg	Bnd Diff	CI Diff	Avg	Bnd Diff	CI Diff	Avg	Bnd Diff	CI Diff
0.05	0.985	0.048	0.019	0.973	0.094	0.023	0.971	0.133	0.050
0.10	0.980	0.091	0.033	0.946	0.167	0.051	0.935	0.232	0.042
0.20	0.953	0.167	0.076	0.873	0.287	0.087	0.828	0.375	0.104
0.30	0.892	0.231	0.121	0.790	0.376	0.149	0.685	0.306	0.117
0.40	0.862	0.286	0.168	0.707	0.277	0.128	0.575	0.170	0.070
0.50	0.796	0.333	0.175	0.610	0.167	0.059	0.484	0.099	0.027
0.60	0.739	0.208	0.112	0.524	0.101	0.034	0.412	0.059	0.020
0.70	0.662	0.126	0.067	0.461	0.059	0.024	0.353	0.035	0.017
0.80	0.600	0.069	0.042	0.403	0.032	0.021	0.311	0.018	0.011
0.90	0.543	0.029	0.033	0.363	0.013	0.014	0.275	0.008	0.007
1.00	0.495	0.000	0.004	0.331	0.000	0.05	0.249	0.000	0.008

Table 1 shows that the difference between the upper and lower limits formula bounds can reach as high as 0.375, for 4 threads and a normalized aggregate loading of 0.20. However, the measured CPU availability is for this same case is much smaller, around 0.104. The difference of the formula-based bound is more precise when the CPU is lightly or heavily loaded.

Table 2: Average measured CPU availability (Avg.) for threads in a quad-core machine along with differences of bound formulas (Bnd Diff) and 90% confidence interval limits (CI Diff).

L/n	8 Thread CPU Availability			12 Thread CPU Avail.			16 Thread CPU Avail.		
	Avg	Bnd Diff	CI Diff	Avg	Bnd Diff	CI Diff	Avg	Bnd Diff	CI Diff
0.5	0.973	0.080	0.015	0.971	0.121	0.017	0.969	0.158	0.013
0.10	0.971	0.150	0.019	0.974	0.216	0.022	0.948	0.273	0.058
0.20	0.973	0.260	0.064	0.959	0.355	0.064	0.862	0.429	0.128
0.30	0.956	0.345	0.153	0.904	0.452	0.121	0.690	0.363	0.149
0.40	0.903	0.413	0.322	0.752	0.357	0.085	0.543	0.225	0.116
0.50	0.768	0.466	0.304	0.604	0.246	0.102	0.411	0.152	0.107
0.60	0.595	0.345	0.299	0.490	0.178	0.081	0.344	0.109	0.079
0.70	0.554	0.264	0.222	0.429	0.134	0.073	0.321	0.081	0.057
0.80	0.493	0.208	0.217	0.358	0.104	0.069	0.268	0.062	0.028
0.90	0.446	0.167	0.146	0.328	0.083	0.079	0.240	0.049	0.029
1.00	0.419	0.136	0.114	0.291	0.067	0.058	0.222	0.040	0.023

Similarly, for a quad core processor, Table 2 shows that the difference between the upper and lower limits formula bounds can reach as high as 0.429, for 16 threads and a normalized aggregate loading of 0.20. However, the measured CPU availability is for this same case is much smaller, around 0.128. The difference of the formula-based bound is more precise when the CPU is lightly or heavily loaded for quad-core processor as well.

5. Conclusion and Future work

This paper developed analytical models (and conducted empirical studies) for predicting (and measuring) CPU availability of JVMs supported by single- and multi-core architectures. As would be expected, degradation in CPU availability occurs when total CPU loading is greater than the total capacity of all CPU cores. In addition to total CPU loading, the total number of concurrent threads is a factor in predicting CPU efficiency; more threads generally incur more context switching overhead, which results in degraded availability. When the total load is less than the total capacity of all cores, the relative alignment of the working and sleeping phases of the threads can have a significant impact on CPU availability. Specifically, increased overlap of the work phases implies lower availability. It was demonstrated that shifting the relative phasing of the threads to reduce possible work phase overlap can improve the performance (i.e., CPU efficiency). Random aggregate load and phase shift values for concurrent threads were assigned for each for an extensive number of experimental measurements. A thread availability scatter plot provides a clear visualization of measured performance based on the density of the dots in the plot. These empirically measured availability values are shown to generally fall within theoretically derived upper and lower bound formulas. A 90% confidence interval for measured availability is shown to provide significantly tighter upper and lower limits than the theoretically derived formulas for upper and lower bounds.

6. References

- [1] Martha Beltrán, Antonio Guzmán and Jose Luis Bosque, "A new CPU Availability Prediction Model for Time-Shared Systems", *IEEE Computer*, Vol 57, July 2008.
- [2] Bill Venners, "Inside the Java 2 Virtual Machine", Thread Synchronization, URL: <http://www.artima.com/insidejvm/ed2/index.html>
- [3] Y. Zhang, W. Sun, and Y. Inoguchi, "Predicting running time of grid tasks on cpu load predictions", *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pp. 286–292, September 2006.
- [4] "Java thread performance", The behavior of Java threads under Linux NPTL, Amity Solutions Pty Ltd – Version 1.5, July 2, 2003, URL: http://www.amitysolutions.com.au/documents/NPTL_Java_threads.pdf
- [5] Ken Arnold and James Gosling, *The Java Programming Language*, Fourth Edition, Addison Wesley, 2005.
- [6] Vitaly Mikheev, "Switching JVMs May Help Reveal Issues in Multi-Threaded Apps", May 2010, <http://java.dzone.com/articles/case-study-switching-jvms-may>.
- [7] Analysis of Multithreaded Architecture for Parallel Computing, Rafael H. Saavedra, David E. Culler, Thorsten Eicken, *2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, 1990.

SESSION
EVALUATION METHODS AND PERFORMANCE
ANALYSIS

Chair(s)

TBA

Examining Anomalous Network Performance with Confidence

Bradley W. Settlemyer, Stephen W. Hodson, Jeffery A. Kuehn and Stephen W. Poole

Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
{settlemyerbw,hodsonsw,kuehn,spoole}@ornl.gov

Abstract—*Variability in network performance is a major obstacle in effectively analyzing the throughput of modern high performance computer systems. High performance interconnection networks offer excellent best-case network latencies; however, highly parallel applications running on parallel machines typically require consistently high levels of performance to adequately leverage the massive amounts of available computing power. Performance analysts have usually quantified network performance using traditional summary statistics that assume the observational data is sampled from a normal distribution. In our examinations of network performance, we have found this method of analysis often provides too little data to understand anomalous network performance. In particular, we examine a multi-modal performance scenario encountered with an Infiniband interconnection network and we explore the performance repeatability on the custom Cray SeaStar2 interconnection network after a set of software and driver updates.*

Keywords: High Performance Computing, Networking, Performance, Benchmarks

1. Introduction

High speed interconnection networks, such as Infiniband and Cray's SeaStar2, offer a multitude of network transmission modes and features. Options such as zero-copy remote direct memory access (RDMA) and congestion avoidance routing make understanding an application's network behavior quite difficult. In particular, the performance of collective communication primitives, such as the Message Passing Interface's MPI_Allreduce, is often dictated by the slowest communicating member of the collective; however, even for point-to-point nearest-neighbor communication patterns, the ability to advance a time stepping solution is typically constrained by the slowest neighbors, not the fastest neighbors. In both cases, a single member of the collective communication that is experiencing poor performance will often severely limit the performance of the entire scientific application. In this paper, we use Confidence to examine poor communication performance due to anomalous network behavior – and in one case, we are able to reconfigure the networking stack to improve network latency.

The Confidence toolkit [1] seeks to present information about the performance of a benchmark or micro-benchmark over the entire range of observed results by presenting an

empirically derived probability distribution that describes the performance of the system under study. Note that the resultant probability distribution only describes the system performance at the time of the benchmark execution. Different systems will have different probability distributions, and the probability distributions may change over time depending on the degree and nature of resource sharing. On the other hand, if the benchmark execution adequately samples the entire network topology, we can expect that a high degree of measurement stability (e.g. stationarity and ergodicity) will exist.

1.1 Related Work

Previous studies have demonstrated the utility of probability distributions (rather than simple summary statistics) to discover multi-modal performance characteristics in computer networks [2], [3]. Our studies further demonstrate that the distributions of high performance interconnect messaging is often skewed with multiple modes present [1]. Summary statistics assume the underlying data is sampled from a normal distribution and typically stress the performance mean which may not be relevant to application performance.

The rigorous evaluation of network performance in HPC systems has been an area of interest for several years. Petrini, et al., described the performance of the Quadrics interconnection network using a suite of network tests [4]. In general, the analysis focused on looking at average and best-case performance measurements on a quiesced system. The HPC Challenge (HPCC) Benchmark Suite extended this approach to include tests on all processing an HPC system [5], both in a pair-wise fashion, and in process rings. These techniques have been used to describe the network (and overall system) performance of the Cray XT4 and Blue Gene/P at Oak Ridge National Laboratory [6], [7], the Blue Gene/L systems at Argonne and Lawrence Livermore National Laboratory [8], and the Roadrunner system at Los Alamos National Laboratory [9].

Bhatel  and Kal  examined the effects of contention in high performance interconnection networks [10]. A benchmark was constructed to have all pairs of processes send messages at the same time with the number of hops between each sender fixed. The results were averaged and reported for each factor of 4 message size between 4 Bytes and 1 MiB and indicated that contention on the network could severely degrade communication performance.

2. Confidence Configuration

Confidence relies on *data binning* to gain insight into the behavior of benchmarks. Data binning is a data pre-processing technique that attempts to quantize observed values into a discrete number of buckets that adequately represent and emphasize the magnitude of the real observed values. By binning millions or billions of benchmark measurements, we can use the resulting frequency distribution to approximate the continuous stochastic process that describes the benchmark performance.

For any benchmark elapsed time measurements may differ by nanoseconds in terms of the returned values, but the timers used to observe those durations are not capable of providing measurements with such high degrees of precision. Confidence includes a measurement abstraction layer called the Oak Ridge Benchmarking Timer, or *ORBTimer*, that both selects an appropriate timer for the underlying hardware (e.g. the Pentium/x86 cycle counter) and calibrates the timer in a manner consistent with the behavior of the actual timed kernel.

Once the fidelity of the timer has been determined, it is straightforward to determine an appropriate number of data bins and a bin width for the benchmark under study. Confidence provides both fixed width data bins and logarithmically scaled bin sizes. The fixed bin width, f_w , is determined by dividing the maximum histogram time, T , by the number of requested bins, C :

$$f_w = T/C \quad (0 \leq i < C). \quad (1)$$

Logarithmically scaled data bins are useful when the timing data varies by several orders of magnitude or the amount of system memory for storing measurements is constrained. To use logarithmically scaled bins, the user must specify a bin size, S that is greater than 0. The logarithmic data bin width, l_w , is described by the following function:

$$\begin{aligned} l_{w_0} &= S \\ l_{w_i} &= e^{S^*i} - 1 \quad (0 < i < C). \end{aligned} \quad (2)$$

2.0.1 Latency Benchmark

For these tests we used *CommTest3*, a network benchmark included with Confidence. For each trial, *CommTest3* performed a pairwise MPI_Sendrecv between every MPI process running as part of the benchmark. MPI_Sendrecv was selected as the benchmarking kernel operation because it was well supported on all platforms and does not subdivide the communication over several user space calls (such as MPI_Wait), which would make it difficult to measure the constituent communication portions.

Figure 1 illustrates the simple micro-benchmarking kernel used in these experiments. All of the processes cycled through each of their possible peers and performed a pairwise MPI_Sendrecv operation of 1 byte of data. The results of these operations were reported in three different ways:

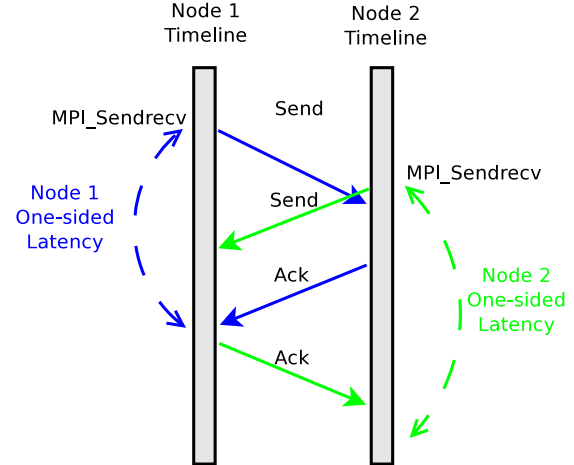


Fig. 1: Pair-wise communication pattern

the latency of the one-sided communication on node1, the latency of the one-sided communication on node2, and the pair-wise communication latency, which is the average of the two one-sided latencies.

2.0.2 Data Analysis

By binning many millions of timing samples, Confidence is able to construct an empirical approximation of the probability distribution of the timing data. The originating random process that generates the values is continuous but the individual measurements are discrete, so we must use a large number of discrete measurements to approximate the continuous probability distribution of the timing data. The resulting frequency distribution is used to mathematically construct the empirical PDFs, and empirical CDFs. Recall that a PDF, f , defines the probability for a random variable, X , to take a value in some range $[a : b]$, as:

$$P[a \leq X \leq b] = \int_a^b f(x) dx. \quad (3)$$

Additionally, Confidence extracts the minimum observation for each pairwise cycle, and bins that data and constructs distribution information and summary statistics for the observed minimums. Although the sample size of the observed minimums is small (1 observation per cycle per host), the measurements are useful as a proxy for the performance of the underlying hardware. In order to increase the accuracy of our minimums distribution we increased the number of benchmarking cycles to 100.

3. Diagnosing an Infiniband Performance Anomaly

We encountered the surprising multi-modal performance shown in figure 2 while examining the latency of pairwise

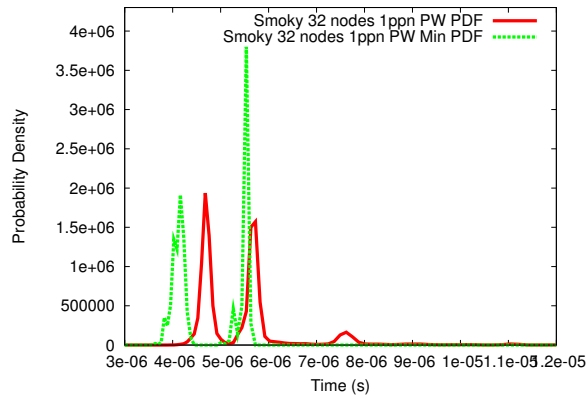


Fig. 2: All pairwise communications and minimum pairwise communications PDF for 32 nodes of Smoky with 1 process communicating per node.

MPI-based communication on the Smoky Commodity Cluster, an Infiniband-based cluster at Oak Ridge. Smoky was an 80 node Linux test and development cluster available at Oak Ridge National Laboratory's National Center for Computational Science (NCCS). Each node contained four 2.0 GHz AMD Opteron processors, 32 GiB of main memory, an Intel Gigabit Ethernet NIC, and a Mellanox Infinihost III Lx DDR HCA. The Infiniband network was switched with a single Voltaire DDR Infiniband Grid Director 2012 using four sLB-2024 24-port Infiniband Line cards. The switch provided 11.52 Tbps of bisection bandwidth with a reported port-to-port latency of 420 nanoseconds. The nodes ran Scientific Linux SL release 5.0, a full Linux operating system based on the popular Red Hat Linux distribution. The benchmark was built using the Portland Group International compiler version 10.3.0 and OpenMPI version 1.2.6.

The pairwise message send latency PDF indicates the presence of three performance modes when 32 Smoky nodes are communicating simultaneously with only 1 communicating process per node. The pairwise minimum distribution shown on the same graph indicates that two of the performance modes are present in hardware-only measurements. The third, much smaller, performance mode may be due to overhead in the network software stack, or it may simply be due to OS-based interrupts. The first performance mode is centered at $4.6\mu\text{s}$, the second mode is centered at $5.7\mu\text{s}$, and the third mode is centered at $7.5\mu\text{s}$. With the aid of Confidence, our goal was to identify if any hardware issues contributed to the performance modes.

3.1 Ensuring Measurement Validity

Figure 3 shows a histogram of the observed timer overheads during our benchmarking run. Although the full version of Linux in use on Smoky results in somewhat noisy timing data, 99.8% of the timing overheads fall into the first

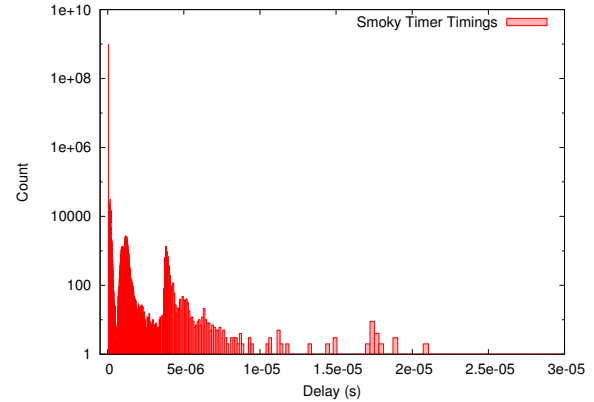


Fig. 3: Histogram of binned x86 timer values on Smoky.

histogram bucket, which spans 0 - 50 nanoseconds, indicating that the timer overhead was most often a negligible component of our measurements.

3.2 Examining Hardware Performance

Confidence decomposes benchmark execution iterations into cycles. For each benchmarking cycle, 200,000 messages are passed between each node-pair. In addition to recording the latencies for the 200,000 messages sent between each host pair, confidence also separately bins the absolute minimum observation for each host pair during a cycle. This best case minimum can be thought of as the actual hardware induced networking latency. We use the pairwise minimums rather than the one-sided minimums, because the one-sided minimum observation is typically only the time it takes to retrieve data from local memory after a successful RDMA put operation. Although the pairwise minimum is likely larger than the actual network hardware overheads, we believe that the pairwise timing acts as an accurate proxy for the actual hardware costs, and is closely correlated with the hardware costs. In order to increase the accuracy of our minimums distribution we increased the number of benchmarking cycles from 10, to 100.

3.3 Analyzing Switching behavior

The switch documentation indicated that within a single line card, each application specific integrated circuit (ASIC) could communicate with every port within the ASIC without an additional network hop. However, there was no communication between the two ASICs that populated each line card. Figure 4 confirms that within the same ASIC, minimum pairwise latencies tended about $3.9\mu\text{s}$, whereas ASIC spanning communications required $4.1\mu\text{s}$ on average.

We hypothesized that hopping across the switch backplane from one ASIC to the next within the switch may not require as many switch hops as crossing between both switch line cards and ASICs. Figure 5 shows the resulting latency PDF

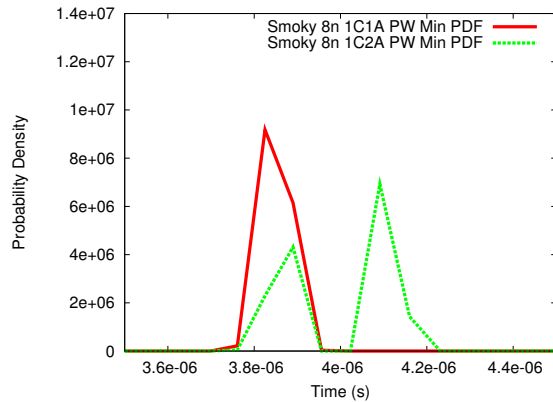


Fig. 4: Pairwise communication minimums for 8 nodes. The first plot engages only a single ASIC; the second plot spans both line card ASICs.

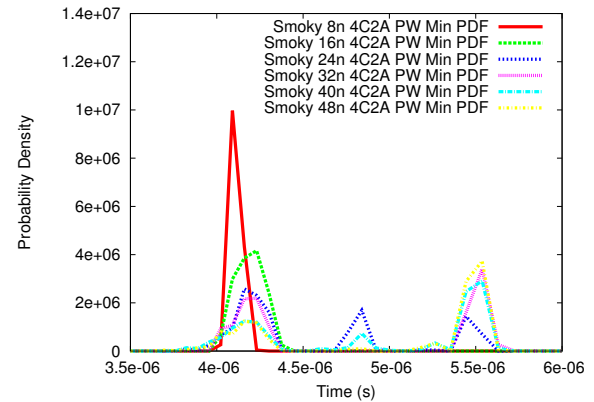


Fig. 6: Communication minimums for nodes distributed evenly across all of the switch resources (line cards and ASICs). Each plotted PDF shows the addition of exactly one node to each ASIC in the switch.

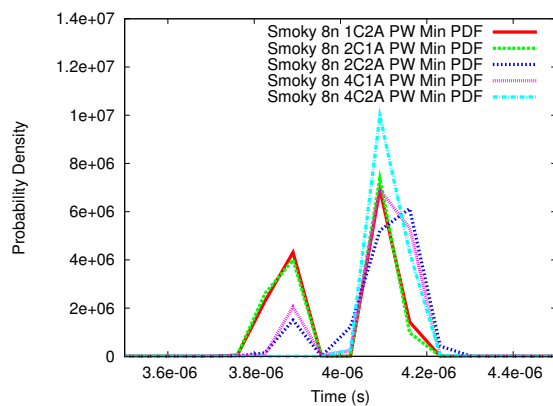


Fig. 5: Communication minimums for 8 nodes distributed in each balanced configuration across line cards and ASICs. The key indicates the number of line cards and ASICs per line card in use for each plotted PDF.

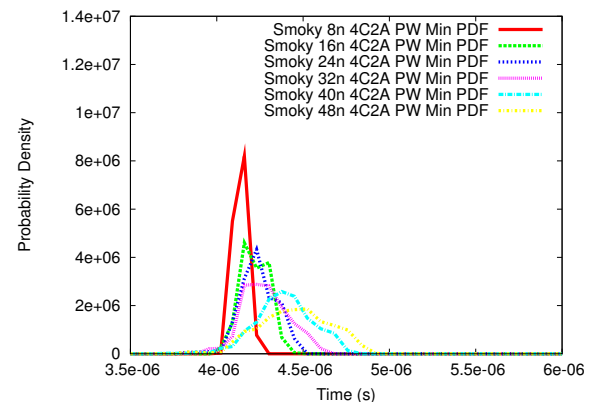


Fig. 7: Communication minimums for nodes distributed evenly across all of the switch resources (line cards and ASICs) with RDMA eager mode enabled for 80 hosts.

for our Confidence jobs that allocated 8 total nodes in each of the following configurations: on a single line card to a single ASIC, one line card using both ASICs, two line cards using a single ASIC per card, two line cards with both ASICs in use, four line cards with a single ASIC in use on each, and four line cards with two ASICs in use per line card. The two resulting performance modes indicated that the switch backplane does not introduce any additional latency, and that the reduced performance detected in our original tests was not likely due to any anomalous switching behavior.

Figure 6 demonstrates the resulting performance as we added nodes to our tests while balancing the results across each switch line card and ASIC. Each line in the plot is the result of adding a single node for each ASIC, and then running the confidence benchmark with that configuration. Here we see the performance modes were not due to switch

hop counts, but instead due to an anomaly that appears at any time more than 16 switch ports were in use.

Further investigation of the issue indicated that the Infiniband driver compiled into OpenMPI used different message send protocols depending on the number of hosts configured in the system. The first 16 hosts an MPI process communicates with use an RDMA eager protocol; however, due to a conservative implementation decision (i.e. concerns about excessive polling costs) all subsequent MPI Sends resort to an eager send protocol that uses an operating system buffer (requiring context switches). Within OpenMPI we were able to adjust the eager RDMA limit to 80 hosts, with figure 7 showing the resultant performance PDFs.

Figure 8 shows the resulting CDF when scheduling 32 processes on Smoky with forced eager RDMA communications versus the default configuration. In addition to the

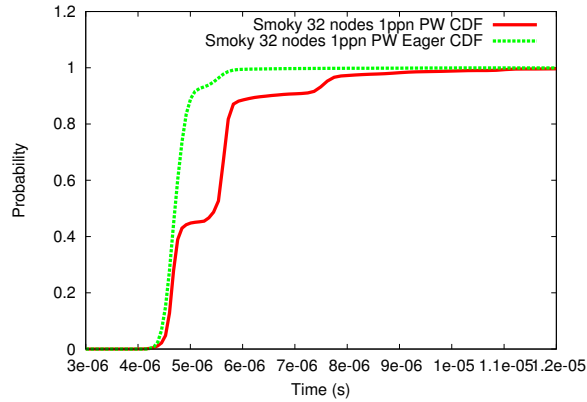


Fig. 8: Before and after pairwise communication costs

improved performance, the removal of the operating system assisted send possibly impacted the smaller performance mode. This may be due to the reduced likelihood of computational noise due to the smaller amount of time spent performing communications per benchmarking run.

4. Assessing the Impacts of Network Upgrades

In our earlier study of the Jaguar network [1], we noted that network latencies were severely impacted by the number of node processes simultaneously sending. While re-performing a series of experiments, we learned that our benchmark results varied greatly from our earlier observations. We noted that the compilers and MPI platform had experienced revisions since our earlier tests, but were skeptical that the software stack had caused the large degree of change we observed in our Confidence-based benchmarking.

Jaguar [11], a Cray XT5, was composed of 18,688 dual socket compute nodes running Compute Node Linux, a lightweight Linux-based operating system. Each socket contained a hex-core AMD Opteron 2435 processor at 2.6 GHz for a total core count of 224,256, and each node included 16GiB of DDR2-800 main memory for a total system memory of 299 TiB. Each node in Jaguar was connected using a SeaStar2 router capable of transmitting 76.8 Gbps in each direction on the 3-dimensional torus network.

Our original benchmark code was built using the Cray XT5 compiler wrapper and MPI libraries, based on the Portland Group International compiler version 9.0.4 and XT Message Passing Toolkit 3.5.1. The more recent configuration relied on Portland Group International compiler version 10.3 and the XT Message Passing Toolkit 4.0.0. All benchmark runs were performed on 64 node allocations randomly selected by the scheduler. The allocations included no more than one shelf from each of 5 separate cabinets, ensuring the allocation spanned all 3 dimensions of the torus network.

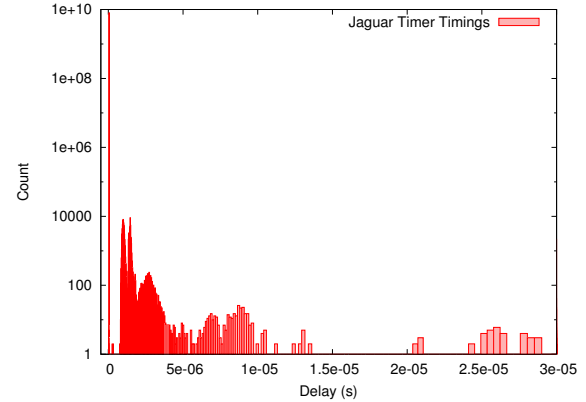


Fig. 9: Histogram of binned x86 timer values on Jaguar

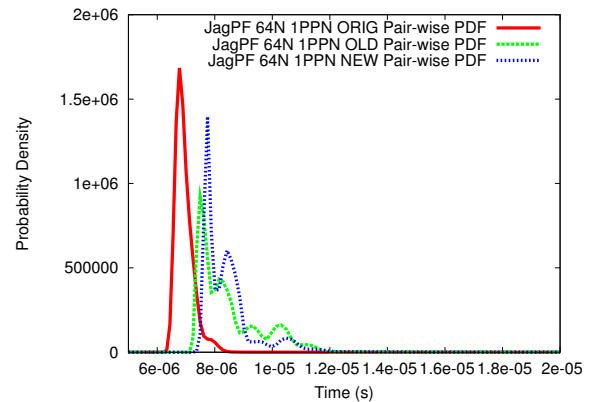


Fig. 10: Pairwise communication latency PDF for 64 Jaguar nodes with 1 communicating process per node. The first line is the original data, the second line uses the old software stack running after system updates, and the third line uses the new software stack running after system updates.

4.1 Validating the Timer

In figure 9 we see a histogram of the system timer measurements for Jaguar. The observed timer skews appear very similar to the OS noise described for the platform [12], and does not appear normally distributed. Over 99.99% of the timer observations fell into the first data bin, which spans 0 - 50 nanoseconds (the total number of timer observations was 8.064×10^9). Recall that these initial timer values (and all of the other gathered values) are reduced by exactly the amount of the *minimum* timer delay observed during the timer calibration phase.

4.2 System Upgrade Measurements

Figures 10 and 11 show the empirical PDF and CDF for a single communicating process per node using all three of our test configurations. The lines labeled “ORIG” are the original observations from four months ago, the lines

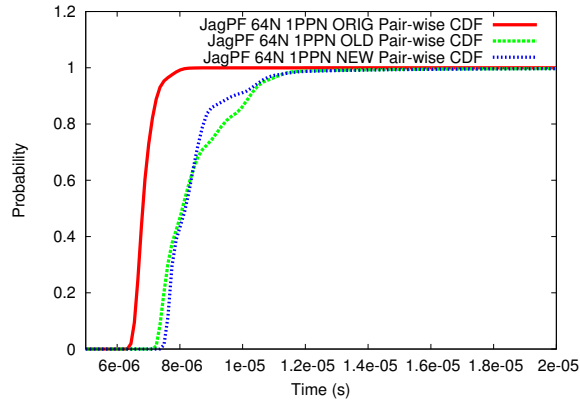


Fig. 11: Pairwise communication latency CDF for 64 Jaguar nodes with 1 communicating process per node.

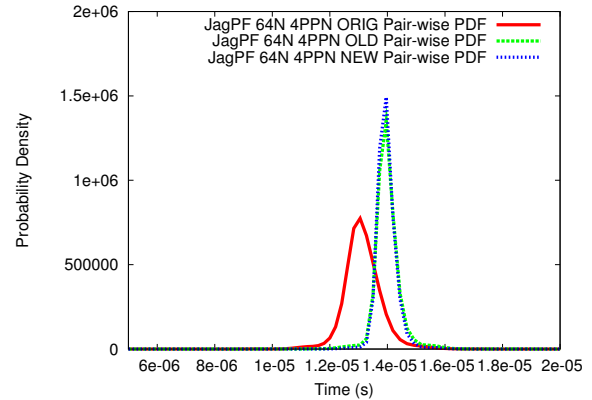


Fig. 13: Pairwise communication latency PDF for 64 Jaguar nodes with 4 communicating processes per node.

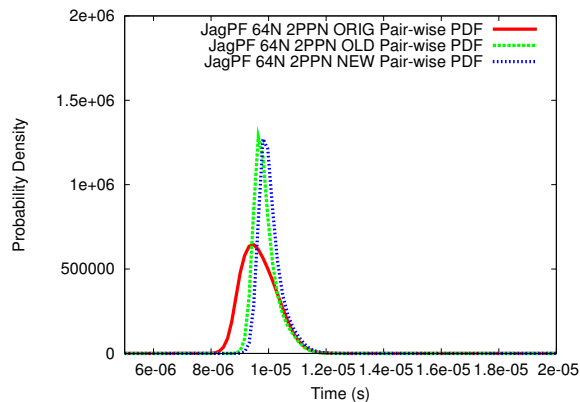


Fig. 12: Pairwise communication latency PDF for 64 Jaguar nodes with 2 communicating processes per node.

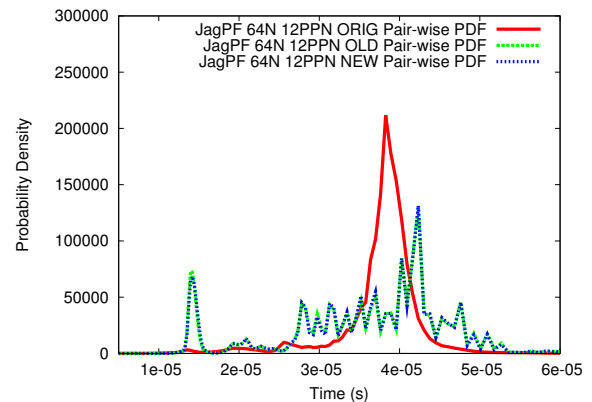


Fig. 14: Pairwise communication latency PDF for 64 Jaguar nodes with 12 communicating processes per node.

labeled “OLD” use the same compiler and parallel tool platforms as the original observations but were observed more recently, and the lines labeled “NEW” were recently observed and used the updated compiler and parallel tool platforms. Both the PDF and CDF clearly show that all of the performance changes are not due to incrementing the compiler and network middleware. Although it appears that some performance changes are due to the change in software configuration, the fundamental changes in the network latency originate outside of the software stack. In collaboration with the Jaguar system administration team we learned that a network driver upgrade had occurred in the intervening period, and that was likely the source of our observed performance differences.

Our original study of the Jaguar network focused on determining the optimal number of independent message originating processes (e.g. MPI tasks) to use in pairing with the Cray XT hardware. In figures 12 and 13 we present the updated PDFs for two and four communicating processes

per node, respectively. We again note that network latency sensitive applications may be well served to use a single MPI task for remote communications and employ a threading approach, such as OpenMP, to leverage the large number of processing cores with a Jaguar compute node. However, it does appear that the network driver update results in a more reproducible (i.e. “peakier”) message latency, even if the performance is slightly degraded from the results of our original measurements.

Figures 14 and 15 show the empirical PDF and CDF for network latency with twelve communicating processes per node. With all node processes sending and receiving network messages it is apparent that the driver update has dramatically altered the measured network latency performance without significantly modifying the mean or median network latency. The CDF clearly demonstrates that in all configurations, observations will be evenly distributed about 38 microseconds. However, after the driver update observed network latencies are much more uniformly distributed over

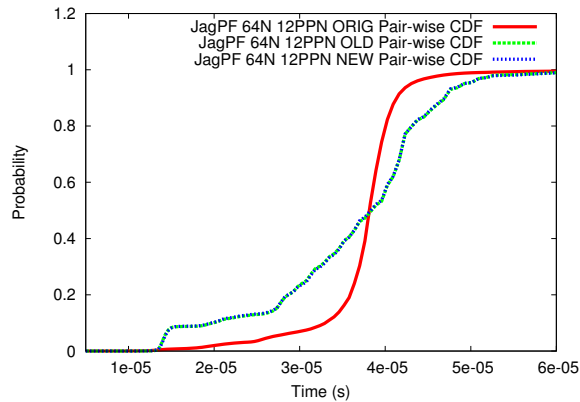


Fig. 15: Pairwise communication latency CDF for 64 Jaguar nodes with 12 communicating processes per node.

the sample space. The original network configuration appears to provide much more predictable network performance (i.e. greater peakiness) with basically identical average case performance. In particular, we expect that collective communication patterns that are performance limited by the slowest participating process will be negatively impacted by the updated performance distribution.

Although the SeaStar driver update clearly impacts the network message latency, it is less clear how the update will affect existing applications. A single communicating process per node will now likely achieve higher latencies and incur greater variability in its measured send-receive latency (though perhaps with less chance of encountering a system halting deadlock). It appears the Cray engineers have endeavored to improve the consistency of network operations when 2 or 4 processes are communicating per node at the cost of increasing the observed network latency. In the case of 12 simultaneously communicating processes, updating the network driver has degraded the send-receive latencies, and has likely decreased the repeatability of network injections.

5. Conclusion

The empirical probability distributions generated by Confidence can aid in measuring performance, locating areas of degraded performance, and evaluating how system component performance changes over time. In this paper we used Confidence to examine anomalous network performance on two separate HPC platforms. In the future we are interested in non-central moments of probability distributions. In HPC systems mean and median performance are not the desired performance levels, thus calculating the distribution moments about the observed minimums may provide a higher quality performance summary than the typical central moments (mean, variance, skew, kurtosis).

Acknowledgments

This work was supported by the Department of Defense (DoD) and used resources at the Extreme Scale Systems Center, located at Oak Ridge National Laboratory (ORNL) and supported by DoD. This research also used resources at the National Center for Computational Sciences at ORNL, which is supported by the U.S. Department of Energy Office of Science under Contract No. DE-AC05-00OR22725. Special thanks to Pawel Shamis for explaining various details related to the OpenMPI BTL OpenIB driver.

References

- [1] B. W. Settlemyer, S. W. Hodson, J. A. Kuehn, and S. W. Poole, "Confidence: Analyzing performance with empirical probabilities," in *Proceedings of 2010 Workshop on Application/Architecture Co-design for Extreme-scale Computing (AAEC)*, September 2010.
- [2] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of internet flow rates," in *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '02. New York, NY, USA: ACM, 2002, pp. 309–322. [Online]. Available: <http://doi.acm.org/10.1145/633025.633055>
- [3] S. Katti, D. Katabi, C. Blake, E. Kohler, and J. Strauss, "Multiq: automated detection of multiple bottleneck capacities along a path," in *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, ser. IMC '04. New York, NY, USA: ACM, 2004, pp. 245–250. [Online]. Available: <http://doi.acm.org/10.1145/1028788.1028820>
- [4] F. Petrini, E. Frachtenberg, A. Hoisie, and S. Coll, "Performance evaluation of the quadrics interconnection network," *Cluster Computing*, vol. 6, no. 2, pp. 125–142, 2003.
- [5] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, and D. Takahashi, "Introduction to the hpc challenge benchmark suite," Tech. Rep., 2005.
- [6] S. R. Alam, J. A. Kuehn, R. F. Barrett, J. M. Larkin, M. R. Fahey, R. Sankaran, and P. H. Worley, "Cray XT4: an early evaluation for petascale scientific simulation," nov. 2007, pp. 1–12.
- [7] S. R. Alam, R. F. Barrett, M. Bast, M. R. Fahey, J. A. Kuehn, C. McCurdy, J. Rogers, P. C. Roth, R. Sankaran, J. S. Vetter, P. H. Worley, and W. Yu, "Early evaluation of IBM BlueGene/P," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.
- [8] K. Davis, A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, S. Pakin, and F. Petrini, "A performance and scalability analysis of the BlueGene/L architecture," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 41.
- [9] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: the architecture and performance of Roadrunner," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.
- [10] A. Bhatele and L. V. Kale, "An evaluative study on the effect of contention on message latencies in large supercomputers," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–8.
- [11] H. W. Meuer, E. Strohmaier, H. D. Simon, and J. J. Dongarra, "TOP500 Supercomputer Sites, 34th edition," in *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '09)*, 2009.
- [12] S. Oral, F. Wang, D. A. Dillow, R. Miller, G. M. Shipman, and D. Maxwell, "Reducing application runtime variability on Jaguar XT5," in *CUG-2010*. Edinburgh, UK: Cray User's Group, 2010.

Methodology to predict the performance behavior of shared-memory parallel applications on multicore systems

John Corredor, Juan Carlos Moure, Dolores Rexachs, Daniel Franco and Emilio Luque

Computer Architecture and Operating Systems Department,
University Autònoma of Barcelona, Barcelona, Spain

john.corredor@caos.uab.es

{juancarlos.moure, dolores.rexachs, daniel.franco, emilio.luque}@uab.es

Abstract—With the advent of multicore architectures, there arises a need for comparative evaluations of the performance of well-understood parallel programs. It is necessary to gain an insight into the potential advantages of the available computing node configurations in order to select the appropriate computing node for a particular shared-memory parallel application. This paper presents a methodology to resolve this issue, by constructing a database with behavior information for various representative shared-memory programming structures, and then estimating the application behavior as a combination of these data. These structures are represented by small specific chunks of code called microbenchmarks (μB) based on the study of memory access patterns of shared-memory parallel applications. μBs set is run on each candidate node, and all execution performance profiles are stored in a database for future comparisons. Then, applications are executed on a base node to identify different execution phases and their weights, and to collect performance and functional data for each phase. Information to compare behavior is always obtained on the same node (Base Node (BN)). The best matching performance profile (from performance profile database) for each phase, is then searched. Finally, the candidates nodes performance profiles identify in the match process are used to project performance behavior in order to select the appropriate node for running the application. Resource centers own different machine configurations. This methodology helps the users or systems administrator of data centers to schedule the jobs efficiently.¹

Index Terms—Performance Evaluation, Memory Access Pattern, Behavioral Profile Application, Multicore Systems.

I. INTRODUCTION

A multicore architecture for processors has emerged as a dominant trend in the chip making industry. It has become widespread, as it provides higher performance at lower cost and more efficient use of energy [1]. A multicore, multi-threaded processor demands higher on- and off-chip memory bandwidth and suffers longer average memory access delays despite an increasing on-chip cache size. Tremendous pressures are put on memory hierarchy systems to supply the needed instructions and data in a timely fashion [2].

Different configurations of computing nodes, produce significant differences in the performance of an application. The task of finding the best configuration is very complex, due to the large number of alternatives for setting the individual components. The ascendancy of multicore in general computing further increased this variety, ergo the difficulty of this selection has been increased.

Finding a suitable configuration is a computer engineering problem that has been addressed using different performance evaluation approaches. Analytical approaches have limitations, for example, in terms of their ability to take into account application characteristics or interaction among multiple processors. Cycle accurate simulation is one of the most commonly used approaches for performance evaluation, but it is very time consuming [3].

Generally, current resource center own several machines of different configurations. The proposed approach helps the users or system administrators to schedule the jobs efficiently.

In this paper, we present a methodology to predict the performance behavior of HPC applications at the node level, for different node configurations. The main advantage of this methodology is that the application does not need to be executed in all available configurations. Further, our method does not involve any simulations, which are often very time-consuming.

Our research aims to design a methodology in order to obtain a behavior prediction of a shared-memory parallel application. The methodology associated profile information of computing nodes with information from the application. The profile information is obtained by running microbenchmarks (μBs) or synthetic test programs in all computing nodes. Instead, application is just executed in one computing node.

This work is organized as follows. Section II refers to related works on this field: the state of the art of performance estimation. Section III gives an overview of the Methodology. Section IV presents experimental validation. Section V closes the paper, showing some conclusions and future work.

¹Supported by the MEC-Spain under contract TIN2007-64974

II. RELATED WORK

A large set of benchmarks have been proposed to represent the overall performance of HPC applications. Among others, NAS Parallel [4] and SPEC [5]. The work of Gustafson and Todi [6] is to find the correlation between HINT and the other benchmarks, indicating that HINT is a superset of these benchmarks and then using it in prediction. McCalpin [7] makes a more complete study of the relationship between benchmarks and application performance, but those ideas are not extended to parallel applications.

In [9], authors compare inherent characteristics of the application of interest against the same characteristics for all programs in the standardized benchmark suite. They use the correlation between the set of benchmarks and the entire application. Instead, our approach seek to be much more precise because we divide the application in significant phases and seek behavior performance similarity in each of the performance profiles obtained from μBs for each phase, as will be shown later.

Marin and Mellor [11] show a complex system for the combination and weighting of attributes of applications with the results obtained from single probes, similar to the work that we propose, however they do not use multicore systems nor parallel applications.

Tikir et al. in [12] use a genetic algorithms approach to model the performance of memory-bound computations. They propose a scheme for predicting the performance of HPC applications based on the results of MultiMAPS benchmarks. MultiMAPS is a memory benchmark that accesses a data array repeatedly, but the access pattern is varied in two dimensions: 1) stride and 2) size of the array. Their approach differs from what we propose in this paper in many aspects. The authors require simulating different cache sizes. Our approach doesn't require simulation.

In [13], Yang makes predictions based on observations of steady performance (parallel codes behave in a predictable way after initial short period). Partial sections of the code running on a reference platform are used to predict the total time of execution of the application on a target platform. Our approach considers representative phases of the application, and in order to obtain the performance profile of the target nodes, we select μBs with similar behavior to the phases of the application.

In [14], Sameh et al. present a method for projecting performance of HPC applications on computing nodes, using published data from SPEC CPF2006 and hardware counters of the base machine. This scheme uses a genetic algorithm as a tool to generate a model of Application Performance HPC benchmarks as a function of substitutes or surrogates. The problem here is that SPEC CPF2006 is not enough benchmarks that represent the similarities of all shared memory scientific applications.

III. METHODOLOGY: PERFORMANCE ESTIMATION

Figure 1 illustrates the general outline of our proposed methodology. Our work uses OpenMP shared-memory parallel programming model [15]. We consider a parallel application, several *target* nodes or candidates nodes, and a *base* node or reference node where we run the application.

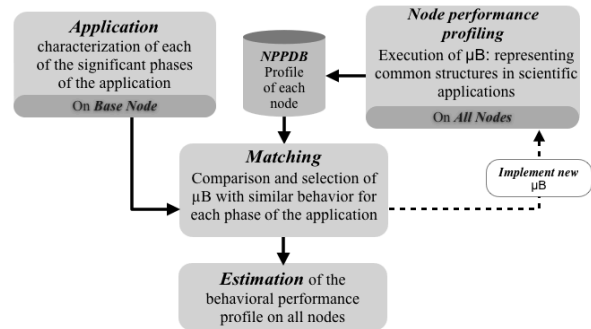


Figure 1: General outline of the methodology

The methodology takes into account a collection of single programs which we call microbenchmarks (μBs). The *Node Performance Profiling* are a collection of characteristics as well as performance numbers on all computing nodes (base and target nodes) given by μBs . The performance numbers are obtained from real hardware execution on all computing nodes. These characteristics along with the performance numbers are then used to build a *Node Performance Profile Data Base (NPPDB)*.

The *application* of interest for which we want to predict performance, is executed only on the *base node*. We obtain the same set of characteristics that we used to build μB for each significant phase and its weight.

Matching process consist of identifying similar *performance profiles* between the *base node* and the information from the application.

Having identified a certain number of *performance profiles* in the *base node* using the *matching process*, the *estimation process* consist of finding the corresponding *performance profiles* of the candidate nodes in the database. Then, the *estimation process* is made to estimate the behavior of the application by combining the *performance profiles*.

The methodology is described in greater detail below.

A. Node Performance Profiling

Benchmarking is the process of running a specific program or workload on a specific machine or system and measuring the resulting performance. This technique clearly provides an accurate evaluation of the performance of that machine for that workload [8]. Microbenchmarks, i.e. very small computational kernels, have used for quantitative measures of node performance in clusters [16].

The *node performance profiling* are obtained by stressing the computing node with synthetic test programs (kernels representative of scientific applications) which are called microbenchmarks “ μB ”, and whose execution time is short [17]. The goal is to characterize the performance of computing nodes against common structures found in different applications. Each executed μB gives *node performance profiles* information of the computing node. In Figure 2 shown the scheme to obtain the *Node Performance Profiling*.

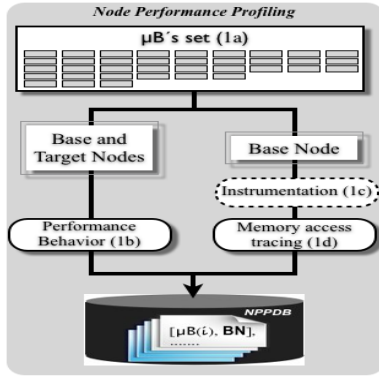


Figure 2: Obtaining the Node Performance Profiling

1) μB s set (1a): First, we implement a set of *microbenchmarks* (μB) that are synthetic test programs, developed in C and OpenMP. Each μB has a set of parameters which characterize the machine and spotlight its strong and weak points. We collect characteristics such as: memory pattern and memory access area (shared or private memory). μB s are designed according to patterns and behaviors previously studied from a set of parallel applications. Table I shows a description of some of the μB s we have used. Input parameters: working size, stride and number of threads.

μB ID	...	$Mbw1(k,s,Th)$	$Mbw3S(k,s,Th)$	$Mbw2R(k,s,Th)$...
Characteristic Memory Access Pattern		a:stride-S sequential	a:stride-1 sequential b:stride-S sequential	a:stride-S sequential b:stride-r random	
Data Type		Double	Double	Double	
Data Density		Dense	Dense	NoDense	
Characteristic Access Private/Private		a(Shared)	a,b(Shared)	a(Private) b(Shared)	
Characteristic Basic Operation		$\sum_{i=0}^k a_i$	$\sum_{i=0}^k a_i \times b_i$	$\sum_{i=0}^k a_i \times b_i$	
Working Set Size		k	k	k	
Stride		s	s	s, r	
Parallelism Level		Th	Th	Th	

Table I: Description of some μB s (will be used in Experimental Validation)

In Table I, $Mbw1(k,s,Th)$ performs one basic operation composed of one read and one accumulation. The data is

read from a vector, with k elements, accessed as a stride-sequential stream. All threads performs the same operations with the same vector. $Mbw3S(k,s,Th)$ has two basic operations: multiply and accumulation, over a and b vectors. Data is read from both vectors, each one with k elements, accessed sequential streams as a stride-1 and stride- s , for stream a and b respectively. $Mbw2R(k,s,Th)$ is similar to $Mbw3S(k,s,Th)$, but (b) vector has stride-random access. a vector is private for each thread, while b vector is shared.

Different executions for different k values and s values is done, in order to be compared with the input parameters of the *Application Characterization*. Th parameter is adjusted to the number of cores of the processor (one thread for core).

We start with a reduced set of reference parallel applications to implement the μB ; but the idea is, to identify new behaviors that are not similar to those found in the *NPPDB*, then we analyze new patterns, and thus implement appropriate new μB s. With each new μB we obtain a new profile and increase knowledge in the *NPPDB*.

2) *Performance Behavior (1b)*: In order to obtain execution time, μB s are executed on all computing nodes (*target* and *base* nodes) for different working set sizes. To get accurate measurements, we run each μB several times.

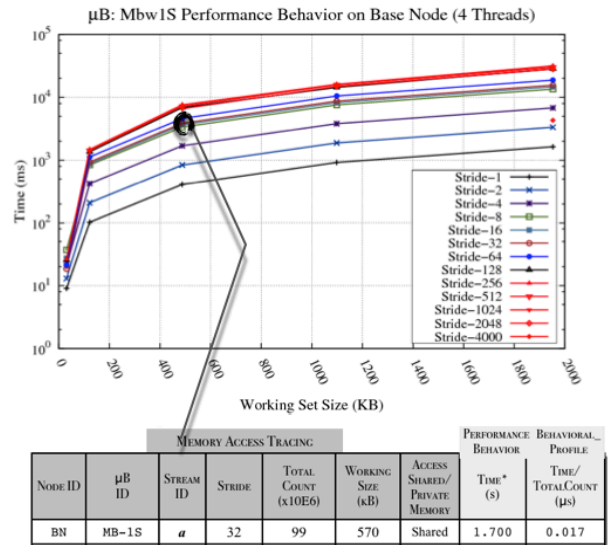


Figure 3: μB : Mbw1S Performance Behavior on Base Node for different stride and working size.

Figure 3 μB : Mbw1S Performance behavior on Base Node for different stride and working size, is shown. X-axis represents working set size (KB) and y-axis represents execution time (ms). The performance behavior of MBw1S μB on *base node* varies with the working set and stride. Lower working size and lower strides, time is better (spatial and temporal locality). These performance behavioral are also found in applications. In the bottom of Figure 3, an example

about information extracted as a performance profile to the desired point.

3) **Node Performance Profile Data Base (NPPDB):** The information about memory access patterns, performance behavior and the behavior profile is stored in *Performance Profile Data Base*.

In order to normalize the results, we calculate the *Behavioral_Profile*: the execution time of μB divide by TotalCount (frequency of stride).

To incorporate new and unknown μB we must extract the characteristics of Table I, in order to obtain the new *performance profiles* of the computing nodes. Then, we must instrumented and get the performance characteristics (Figure 2 (1c) and (1d)) seen in Table I. The new *performance profiles* will be inserted into the *NPPDB*.

B. Application Characterization

The behavior of a program is not random. Researches have shown that it is possible to accurately identify and predict these phases in program execution [18]. Figure 4, depicts the *Application Characterization* scheme.

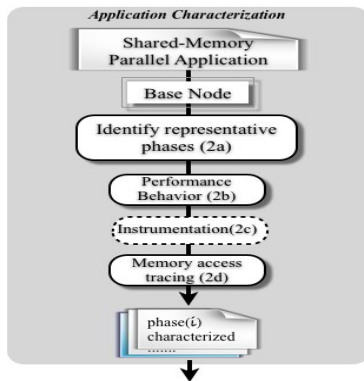


Figure 4: Application Characterization scheme

1) **Identify representative phases (2a):** We need to obtain representative phases of the application, the runtime of each representative phase and the total runtime. To identify each representative phase, we look for OpenMP structures (forks and Join) in the code. Then, analysis is done separately for each phase. We consider as representative phases, those phases whose execution times are meaningful with respect to total running time, for a given input data size. For each representative phase, we need to gather a trace of memory accesses for further analysis.

2) **Performance Behavior (2a):** In order to obtain execution time for each representative phase, the application is executed on the *BN* for a given data input size and their weights.

3) **Instrumentation (2c):** The probes is placed at each identified load/store (identify by streams) to precisely capture the data references issued by each representative phase of

application. The data stream (load/store) is characterized with respect to local data strides [18]. A local stride is defined as the difference in the data memory addresses between temporally adjacent memory accesses come from a single instruction — this is done by tracking memory addresses for each memory operation.

4) **Memory access tracing (2d):** For each memory-access instruction we assign its most frequently used stride along with its stream length. A load or store instruction is modeled as a memory operation that accesses a circular and bounded stream of references, i.e., each memory access walks through an array using its dominant stride value and then restarts from the beginning of the array. To compute *total count* stride, an arithmetic instruction in each basic block is assigned to increment the stride value for the memory walk. The stride value itself is stored in a register.

The tracing process utilizes data structures for fast storage and retrieval. For each stream different parameters are obtained: total count access, stream ID, stride, stride count, lower and higher memory address. We detect whether the vectors are shared, if the data region to which access is the same (memory area) for all threads, otherwise are private.

5) **Phase characterized:** The information of memory access patterns, performance behavior and the behavior profile is placed on *phase characterized*. This process help to find the match with *Node Performance Profile* information from *NPPDB*. Additionally, in order to normalize the results, we calculate the *Behavioral_Profile*: the execution time of phase divide by TotalCount (frequency of stride).

C. Matching Process

It consists of finding the best match between the *node performance profile* in *NPPDB* and *Phase characterized* information. Match start into identify similar memory access patterns between phase application and the *node performance profile*. The idea is to find those *node performance profile* that can represent the behavioral profile applications, or behavioral profile for each representative phase of application. Figure 5 depicts *Matching Process* and *Estimation Process*.

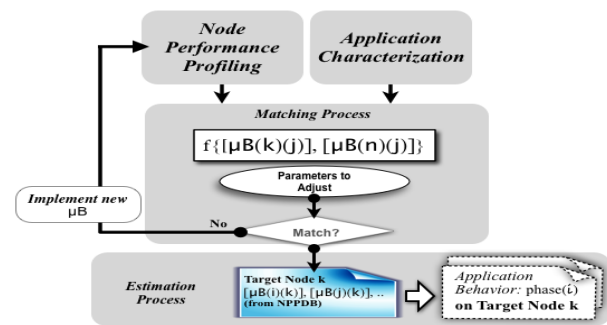


Figure 5: Matching Process and Estimation Process scheme

The performance applications will be a composition of the *Node Performance Profiles*, given by μBs , to estimate performance on the *BN* (where we have also executed the application). The matching process starts to compare the list of characteristics Table I and in priority order. First, we search matching of Memory Access Patterns and so on.

We consider three possibilities for matching:

- In case of similarity between one phase (of the application) and the *node performance profile* on the *BN* (*BN*: Base Node), it may be necessary to adjust the input parameters (stride, working size and number of threads). In order to validate the behavior prediction we compare the values predicted by the methodology and the real ones. The idea is to obtain the best μB match with input parameters tuning (Parameter for Adjustment). This guides the selection of *node performance profile*.
- In case of finding multiples but similar *node performance profiles* with the phase of application, we selected which have the smallest difference error of *Behavioral_Profile(s)* between *Node Performance Profile* and *phase application* characterization.
- In case of not finding similar *node performance profile* on the *NPPDB* with *phase* characterization, it will be necessary to implement new μBs in order to update the μBs set.

D. Estimation Process

Here, we estimate the performance behavior on the target computing node. Up to now, we have divided the program in representative phases, and each phase has obtained a characterization. For each phase, we find the *performance profile* match from μB in *NPPDB* (See III-C). We locate those *performance profile* information from the target nodes (from *NPPDB*, μBs identified previously).

We calculate the behavioral performance of the entire application in (1), sum all the *performance profiles* (of each phase) from the target nodes by their weights.

$$TotalBehavior = \sum_{i=1}^n Profile_i \times weight_i \quad (1)$$

This information (Total Behavior) allows us to compare the performance behavior of the application in different compute nodes and to select the appropriate node.

IV. EXPERIMENTAL VALIDATION

A. Systems: Base Node and Target Nodes

In order to experimentally validate the methodology we used the compute node configurations shown in Table II.

BN, TN1 and TN2 are systems with Intel processors, with a L2 cache shared to each of two cores and a 64-bit Linux O.S. with (x86_64). The target node systems include a dual-socket, and we executed the algorithm and the μBs in all cases on just 4 cores in each node. TN3 is a system with

	BN	TN1	TN2	TN3
Processor	Quad-core Q9400 Intel	Dual-core 5160 Intel (x2)	Quad-core E5430 Intel (x2)	Quad-core 2352 AMD (x2)
Cores	4	4	8	8
On-chip Private	32KB (L1)	32KB (L1)	32KB (L1)	512KB (L2)
On-chip Shared	3MB (x2) (L2)	4MB (x2) (L2)	6MB (x2) (L2)	2MB (L3)
Memory	4 GB	12GB	12GB	4x2 GB NUMA
GHz	1.99	1.99	2.66	2.11
O.S.	x86_64 GNU/Linux	x86_64 GNU/Linux	x86_64 GNU/Linux	x86_64 GNU/Linux
Kernel	2.6.26-2	2.6.16.46	2.6.18-8	2.6.32-19
Compiler	gcc-4.3	gcc-4.3	gcc-4.3	gcc-4.4

Table II: System configurations: Base and Target Nodes

AMD processor with L2 cache private for each core and L3 cache shared in both die, and O.S. with 64bits (x86_64).

Previously, all the μBs are executed on all computing nodes in order to obtain the *Node Performance Profiles* and stored in *NPPDB*.

B. Application Characterization

We used the NPB OpenMP-C benchmark Block-Tridiagonal (BT Class A and B) [4] and N-body simulator, as an example to show our methodology. BT is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. We have identified a recurring phase which appears 3 times. In Table III, the weights and execution time for each phase, is shown. The phases represent around 35% of overall execution time of the application. Once identified, we instrumented this phases in order to obtain the characterization.

	Class A	Weight	Class B	Weight
Total Execution Time (s)	123.68		550.40	
Time phase (s): btA.1	17.63	14.25%	73.42	13.34%
Time phase (s): btA.2	13.01	10.52%	53.92	9.80%
Time phase (s): btA.3	12.88	10.41%	53.47	9.71%
Time Average (s)	14.51		60.27	
Sum 3 phases (s)	43.52	35.19%	180.81	32.85%

Table III: Phase BT (A and B Classes): Weights and execution time of each of the phases and BT total running time.

In Table IV, BT phase characterization is shown. We use average execution time in order to obtain the performance behavior and the behavioral profile metrics.

For N-Body shared-memory parallel application, the total particles is defined by: 640000, 1280000. Total steps and time increment are fixed in 100 times for all cases. We identify a single phase characterized in Table V.

MEMORY ACCESS TRACING							PERFORMANCE BEHAVIOR	BEHAVIORAL PROFILE	
NODE ID	PHASE ID	PATTERN ID	STREAM ID	STRIDE	TOTAL COUNT (x10E6)	WORKING SIZE (kB)	ACCESS SHARED/PRIVATE MEMORY	TIME* (s)	TIME/TOTALCOUNT (μ s)
BN	btA	ph12	a	600	12	4717	Private	14.51	1.209
BN	btA	ph12	a	600	50	18836	Private	60.27	1.205

Table IV: Phase characterized for class A and B. (*We use average executing time of the three representative phases in order to obtain the behavioral profile metrics (See Table III).

MEMORY ACCESS TRACING							PERFORMANCE BEHAVIOR	BEHAVIORAL PROFILE	
NODE ID	PHASE ID	PATTERN ID	STREAM ID	STRIDE	TOTAL COUNT (x10E6)	WORKING SIZE (kB)	ACCESS SHARED/PRIVATE MEMORY	TIME* (s)	TIME/TOTALCOUNT (μ s)
BN	pA	pb42	a	32	55599	390	Shared	881.07	0.0158
BN	pA	pb42	a	32	99900	781	Shared	1611.36	0.0161

Table V: N-Body characterized for 640000 and 1280000 particles.

C. Matching Process

After executing the applications on the *BN*, we start to compare characteristics between each phase of applications (see section IV-B) with each *node performance profiles* on *NPPDB*.

In phase of BT, we look for a *performance profile* from a μB s with stream sequential 600-stride, private memory access, working set 4,7 MB and 18,8 MB respectively. In phase of N-Body algorithm, we look for a performance profile from *NPPDB* with stream sequential 32-stride, shared memory access, working set 390KB and 781KB respectively. Table VII shows *performance profiles* selected to match VI-a and VI-b with BT phase and N-Body phase respectively.

In Table VII, match and error between phases and performance profiles is shown. Once, we select the performance profile of μB MB-1U and MB-1S, but executed on all candidates nodes, in order to obtain the *Behavioral Performance Estimation* for both applications.

D. Estimation Process

Once we have the matching performance profile on *BN* (μB s) we look for the values of these performance profile of the candidate nodes or *TN(i)* from the *Performance Profile Data Base*, and we obtain the behavioral profile estimation of the algorithms on the Target Nodes.

Figures 6-a and 6-b show the Estimation of Performance Behavior given by μB s: MB-1U and MB-1S, to predict performance behavior of phases BT and N-Body applications on all nodes.

To validate the Estimation Performance Behavior obtained previously (Figure 6-a and 6-b), Figures 7-a and Figure 7-b show the real behavior values of the BT and N-Body.

MEMORY ACCESS TRACING							PERFORMANCE BEHAVIOR	BEHAVIORAL PROFILE
NODE ID	μB ID	STREAM ID	STRIDE	TOTAL COUNT (x10E6)	WORKING SIZE (kB)	ACCESS SHARED/PRIVATE MEMORY	TIME* (s)	TIME/TOTALCOUNT (μ s)
BN	MB-1U	a	600	1.1	4708	Private	1.37	1.249
BN	MB-1U	a	600	1.8	18809	Private	2.31	1.279

(-a) Performance Profile of μB MB-1U

MEMORY ACCESS TRACING							PERFORMANCE BEHAVIOR	BEHAVIORAL PROFILE
NODE ID	μB ID	STREAM ID	STRIDE	TOTAL COUNT (x10E6)	WORKING SIZE (kB)	ACCESS SHARED/PRIVATE MEMORY	TIME* (s)	TIME/TOTALCOUNT (μ s)
BN	MB-1S	a	32	99	390	Shared	1.50	0.0152
BN	MB-1S	a	32	199	781	Shared	3.02	0.0154

(-b) Performance Profile of μB MB-1S

Table VI: *Node Performance Profiles* given by μB s a) MB-1U and b) MB-1S on *BN*

WORKING SIZE (kB)	BEHAVIORAL_PROFILE BT (μ s)	BEHAVIORAL_PROFILE μB : MB-1U (μ s)	DIFF (%)
4717	1.209	1.249	3.291%
18836	1.205	1.279	6.106%

(-a) Match between BT phase and MB-1U performance profile

WORKING SIZE (kB)	BEHAVIORAL_PROFILE N-Body (μ s)	BEHAVIORAL_PROFILE μB : MB-1S (μ s)	DIFF (%)
390	0.0158	0.0152	3.797%
781	0.0161	0.0154	4.348%

(-b) Match between N-Body phase and MB-1S performance profile

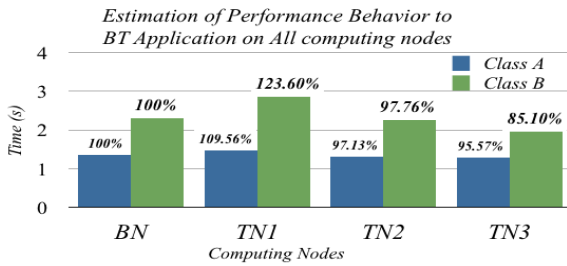
Table VII: Match and error between phases and performance profiles

In Figure 7, the real behavior of both applications is very similar to the estimation behavior given by the methodology. The time to obtain the estimation performance behavior is very significant respect to the time it takes to executing the application on all compute nodes.

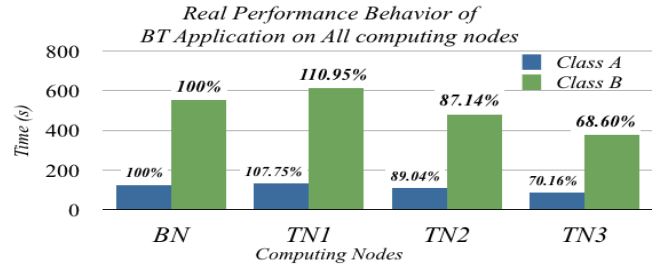
V. CONCLUSION

We have presented a methodology for selecting a suitable multicore system for shared-memory parallel applications by characterizing the behavior of various computing node architectures using μB s based of kernels which are representative of parallel application constructs. In this work we focus in μB s based only on memory access patterns, which are often critical for performance on multicore system. The execution times for all μB s stressing the memory system of four computing nodes, show that they are useful in identifying meaningful performance differences.

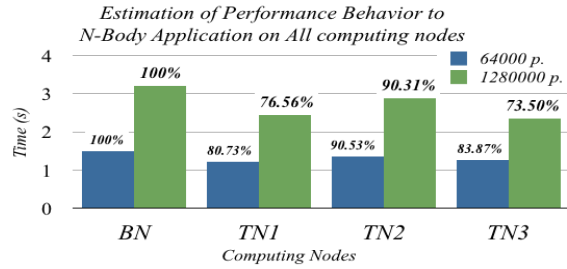
We have outlined a method for application characterization, and have applied the idea on two scientific algorithms, for different input data sizes. The algorithm matches very well with proposed μB s, and thus enables the rapid selection of an appropriate computing node.



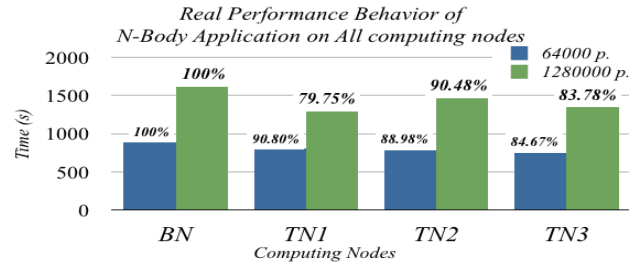
(-a) Estimation Performance Behavior BT by μB MB-1U



(-a) Real Performance Behavior BT Application



(-b) Estimation Performance Behavior N-Body by μB MB1-S



(-b) Real Performance Behavior N-Body Application

Figure 6: Performance Estimation BT and N-Body applications given by μB MB-1U and MB-1S on All computing nodes

Figure 7: Real Performance Behavior BT and N-Body application on All computing nodes

The methodology allow us detected the application behavior on all computing nodes, performance behavioral differences between the computing nodes (to the application) and selecting the suitable computing node for run the application.

REFERENCES

- [1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2007.
- [2] J. Corredor, J. C. Moure, D. Rexachs, D. Franco, and E. Luque, "Active learning processes to study memory hierarchy on multicore systems," *Procedia Computer Science*, vol. 1, no. 1, pp. 921 – 930, 2010, iCCS 2010.
- [3] C. P. Joshi, A. Kumar, and M. Balakrishnan, "A new performance evaluation approach for system level design space exploration," in *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*. New York, NY, USA: ACM, 2002, pp. 180–185.
- [4] D. Bailey, E. Barszcz, D. Browning, R. Carter, L. Dagun, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS parallel benchmarks," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing 1991*. New York, NY, USA: ACM, 1991.
- [5] <http://www.spec.org/cpu2006/>, "Spec cpu2006."
- [6] J. L. Gustafson and R. Todi, "Conventional benchmarks as a sample of the performance spectrum," *J. Supercomput.*, vol. 13, no. 3, pp. 321–342, 1999.
- [7] J. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, 1995.
- [8] R. H. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Trans. Comput. Syst.*, vol. 14, pp. 344–384, November 1996.
- [9] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, ser. PACT '06. New York, NY, USA: ACM, 2006, pp. 114–122.

- [10] U. Krishnaswamy and I. D. Scherson, "A framework for computer performance evaluation using benchmark sets," *IEEE Trans. Comput.*, vol. 49, pp. 1325–1338, December 2000.
- [11] G. Marin and J. Mellor-Crummey, "Cross-architecture performance predictions for scientific applications using parameterized models," in *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2004, pp. 2–13.
- [12] M. Tikir, L. Carrington, E. Strohmaier, and A. Snaveley, "A genetic algorithms approach to modeling the performance of memory-bound computations," in *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.
- [13] L. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 40.
- [14] S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. Taylor, and X. Wu, "Performance projection of hpc applications using spec cfp2006 benchmarks," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.
- [15] OpenMP, "Openmp application program interface." [Online]. Available: <http://www.openmp.org>
- [16] M. Sottile and R. Minnich, "Analysis of microbenchmarks for performance tuning of clusters," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 371–377.
- [17] J. Corredor, J. C. Moure, D. Rexachs, D. Franco, and E. Luque, "Selecting a suitable multicore system for shared-memory parallel application on multicore systems," in *PDPTA 2010: Proceedings of the 2010 International Conference on Parallel and Distributed Processing Techniques and Applications*, CSRA Press, Las Vegas, Nevada, USA, 2010, pp. 228–234.
- [18] J. Lau, S. Schoemackers, and B. Calder, "Structures for phase classification," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on - ISPASS*, 2004, pp. 57 – 67.

Effects of GPU and CPU Loads on Performance of CUDA Applications

M. Bobrov¹, R. Melton¹, S. Radziszowski², and M. Łukowiak¹

¹Department of Computer Engineering, Rochester Institute of Technology, Rochester, New York, USA

²Department of Computer Science, Rochester Institute of Technology, Rochester, New York, USA

Abstract — *General purpose computing on GPUs provides a way for certain applications to benefit from a commonly available massively parallel architecture. As such deployment becomes more widespread, multiple GPU applications will have to execute on the same hardware in systems that have only one GPU. The aggregate loads of the GPU and CPU impact the performance of each application. This work investigates the effects of CPU and GPU loads on the performance of two CUDA GPU applications with significantly different CPU-GPU interaction profiles: implementations of the AES encryption and Keccak hashing algorithms. The percentage degradation in performance of these applications from CPU and GPU loads indicates dependence on the total execution time of the application, with the greatest degradation for the shortest execution times. Performance degradations as high as 22% and 36% were observed for CPU and GPU loads, respectively.*

Keywords: CUDA; GPGPU; GPU; load; performance

1 Introduction

The advent of NVIDIA's Compute Unified Device Architecture (CUDA) and ATI's FireStream Technology has shifted Graphics Processing Units (GPUs) from primarily graphics enabling devices to general purpose stream processing systems. These GPU architectures are a cost effective alternative to traditional parallel processing machines, (e.g., clusters), with comparable performance for certain applications [1]. This change ushers in a new era in computing, which allows any modern personal computer to take advantage of parallel processing capabilities previously available only in specialized systems.

For such applications, processing may occur primarily on the GPU or may be partitioned between the GPU and CPU. The first configuration will efficiently support only a certain class of applications whose computations fit the single program, multiple data (SPMD) paradigm with a sufficient ratio of computations to memory accesses. On the other hand, the second configuration with the workload partitioned between GPU and CPU provides the opportunity for a wider range of applications to benefit from GPU computing by

offloading only the part of the computation that can best benefit from the GPU architecture.

If a CPU/GPU system is not dedicated to execution of an application, performance of that application will be affected by the other applications targeting the same GPU. In other words, there is the potential for additional CPU and/or additional GPU loads. As offloading tasks from the CPU to the GPU on standard desktop configurations becomes more common, the likelihood of having multiple loads from different applications increases. Such is the case for a general desktop user who will not have a dedicated GPU for non-graphics related tasks. Thus, performance in a typical system will be affected by other applications run by the user.

This research investigates the effects of additional CPU and GPU loads on CUDA performance. The Advanced Encryption Standard (AES) and Keccak hashing algorithms were selected as the test cases. CPU and GPU loads were simulated using various applications, and the performance of the encryption and hashing algorithms was recorded. These values were then used to determine the effects of CPU and GPU loads on performance.

2 CUDA

CUDA is a highly parallel computing architecture of recent NVIDIA GPUs [2]. Unlike traditional GPUs, CUDA GPUs are designed with greater focus on data processing as opposed to flow control and caching. They are capable of executing thousands of lightweight threads simultaneously with many more queued. This high degree of parallelism leads to an immense increase in potential performance such that current generation GPUs can vastly outperform contemporary CPUs in certain applications [3]. However, not all applications can realize these benefits. CUDA is based on the stream processing model, which is an extension of the SIMD (single instruction, multiple data) paradigm. This design paradigm makes CUDA optimal for performing a single program instruction many times on different data elements. The rest of this section briefly describes CUDA stream processing, and it follows the presentation in [2–4].

The CUDA architecture consists of two main components: the memory and the processing cores, which

work in conjunction. Their interaction must be considered carefully when designing a CUDA application. Memory is divided into five categories: global, constant, textured, shared, and local. Each type of memory has distinct features with regard to location, caching, and access. The overall architecture is shown in Fig. 1.

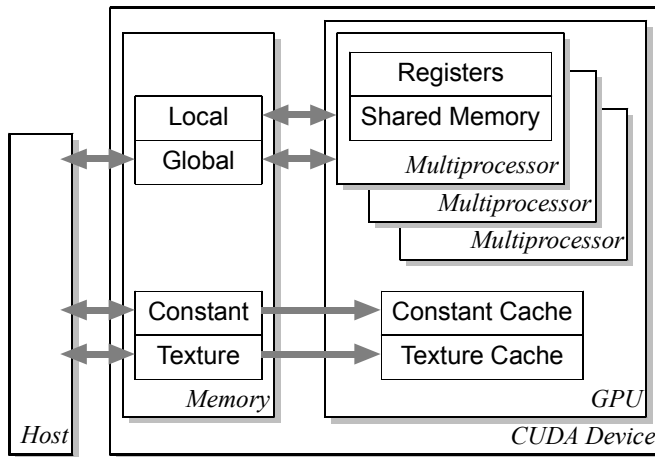


Figure 1. CUDA hardware architecture [4]

The most basic unit of execution in the CUDA architecture is the thread, and threads are organized in a hierarchy, as depicted in Fig. 2. Each thread is allocated a segment of local memory for local variables. Threads may be grouped into 1-dimensional, 2-dimensional, or 3-dimensional blocks, which consist of up to 512 threads per block. Each block has a unique section of shared memory allotted to it, which can be accessed by all threads belonging to that block. All blocks execute independently, but all threads within a block execute simultaneously. There is a mechanism to synchronize threads within a block. At the top level of the thread hierarchy, blocks are grouped into 1-dimensional or 2-

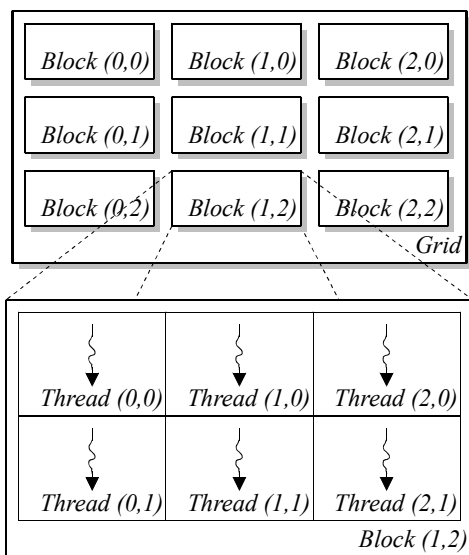


Figure 2. CUDA thread hierarchy [2]

dimensional grids. All grids have access to global memory, constant memory, and texture memory.

To facilitate software design, CUDA implements numerous extensions to ANSI C. Applications are divided into two categories: code designed to execute on the host CPU and code designed to execute on the GPU. The code that is to execute on the GPU is called the kernel. Communication between the CPU and GPU is achieved through memory reads and writes.

CUDA processing consists of four steps, as indicated in Fig. 3: 1) data transfer to GPU memory, 2) CPU invocation of kernel, 3) GPU kernel execution, and 4) data transfer from GPU memory. The first step before executing a kernel is a transfer of data for GPU processing to memory on the GPU. Next, the CPU initiates kernel execution on the GPU. Once execution is complete, the CPU retrieves the processed data from the GPU. Since communication between a CPU and its peripherals is relatively slow, the process of copying data back and forth can often be a major bottleneck. Therefore, an important aspect of an efficient CUDA implementation is the ability to overlap GPU communication from/to the CPU or from/to PC memory with GPU computation.

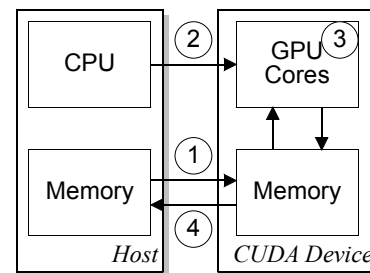


Figure 3. CUDA process flow

3 Test applications

Given the roles of both computation and communication in GPU performance, two general types of algorithms were identified for evaluation of the performance effects of other CPU and GPU loads: communication intensive and computation intensive. A GPU application that is communication intensive requires a significant amount of data transfer from/to the CPU or from/to PC memory, to the extent that execution time is dominated by this communication. In contrast, GPU computation time dominates the execution time of a GPU application that is computation intensive.

Analyzing the performance effects of additional loads on each type of application required suitable candidates. The performance results of GPU implementations of basic cryptographic algorithms [5] provided insight for selecting a test algorithm to represent each type of GPU application. AES was selected as an algorithm whose execution time is dominated more by CPU-GPU communication, and Keccak

was selected as an algorithm whose execution time is dominated more by GPU processing.

3.1 AES

Advanced Encryption Standard (AES) is based on the principles of substitution-permutation networks (SP networks) [6]. To begin, the plaintext to be encrypted is divided into fixed-length blocks of data. These blocks are then converted into a 4x4 array of bytes, known as the state, as illustrated in Fig. 5, where the shading signifies grouping of bytes in columns (words).

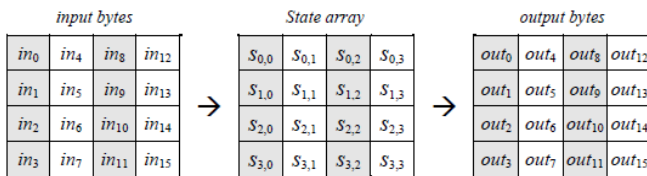


Figure 5. AES state array [6]

Multiple rounds of substitutions, permutations, and key-based operations are performed on the input data to obtain the encrypted ciphertext. The substitution portion of the cipher is a simple replacement of each byte in the array with its entry in a fixed 8-bit Rijndael substitution box (S-box). Next, the permutation portion of the cipher consists of two steps: shift and mix. Shift consists of a row permutation in the form of a left-circular shift, starting with a zero shift for the top row and increasing the shift stride by one for each consecutive row. Mix is then a linear transformation of bytes forming columns of the state matrix. Finally, the round key is determined using Rijndael's key schedule, and the key is then added to the state via a bitwise exclusive or (XOR).

To increase the security and usability of block ciphers like AES, numerous modes of operation have been developed [7]. These modes extend the algorithm in order to ensure that identical message blocks encrypted at different positions in the plaintext with identical keys will not produce equal values.

The tested implementation of AES uses counter (CTR) mode, which performs the AES encryption on a counter value and XORs the result with the corresponding message block to obtain the encrypted output. CTR mode has two characteristics that are favorable for an efficient CUDA implementation. First, it preserves block-level parallelism, which represents the bulk of parallelism available in AES. Second, its encryption of a counter value instead of the plaintext provides the potential for reducing data transfers between the CPU and GPU; the final XOR can be performed either on the GPU or on the CPU.

3.2 Keccak

Keccak is a hash function based on sponge construction [8], and it is one of five finalists in the National Institute of

Standards and Technology (NIST) Cryptographic Hash Algorithm Competition to select SHA-3 [9]. The sponge construction, depicted in Fig. 6, consists of two steps: absorbing and squeezing. It operates on a state, which is arranged in a 5x5 array of 64-bit lanes as shown in Fig. 7.

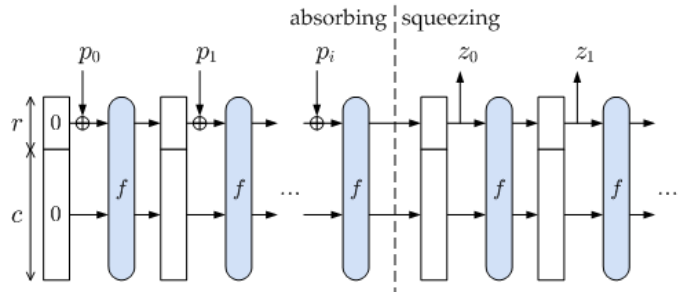


Figure 6. Sponge construction [8]

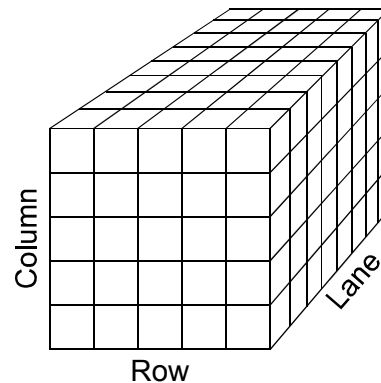


Figure 7. Keccak state [10]

Multiple rounds of squeezing and absorbing are performed on the input data to obtain the final hash. The absorption phase absorbs one r -bit block of the input message at a time by XORing the block with the state and then scrambling the result using a function f . Absorption continues until all blocks of the message have been absorbed. Likewise, the squeezing phase uses the function f to scramble the data further.

The function f used by Keccak is a permutation, which incorporates innovative security improvements [8]. The permutation performed by Keccak can be considered either as an SP network with five-bit wide S-boxes or as a combination of linear transforms followed by a very simple nonlinear transform [11]. The tested implementation of Keccak consists of 24 rounds, which is the recommendation for SHA-3 [8].

4 Test methodology

The test system utilized consists of a dual-core AMD Athlon 5600+ CPU and an NVIDIA GeForce GTX 285 GPU. The GTX 285 contains 240 processing cores and 1 GB of memory. Although this particular model is a midrange GPU, it is generally representative of CUDA enabled GPUs.

For each application, the total execution time (t_{Total}) and the GPU execution time (t_{GPU}) were measured. The total execution time is the time required to encrypt or hash a dataset, including time to transfer data between the CPU and the GPU. The GPU time consists only of the time required to compute the encryption or hashing on the GPU. For each dataset size and application variant, (e.g., AES-128, AES-192, AES-256, Keccak-224, Keccak-256, Keccak-384, and Keccak-512), these timings were measured 1000 times. The average time for each was then calculated.

To measure the effects of CPU and GPU loads additional to the encryption algorithms, CPU and GPU loads were simulated using custom applications. The CPU load application was a simple infinite loop with varying sleep times to achieve the desired 20%, 40%, and 60% loads. The GPU load application consisted of rotating a number of displayed images. Varying the number of images adjusted the GPU load to 25%, 50%, or 75%.

The loads of these applications were determined using Microsoft Perfmon for the CPU and TechPowerUp GPU-Z version 4.4 [12] for the GPU, (shown in Fig. 8). These industry proven tools provide an estimate of the average load over a given period of time. Both applications were run in parallel with the test code. They produced files containing measured CPU loads and GPU loads, respectively. These loads were recorded every second; the average load over the

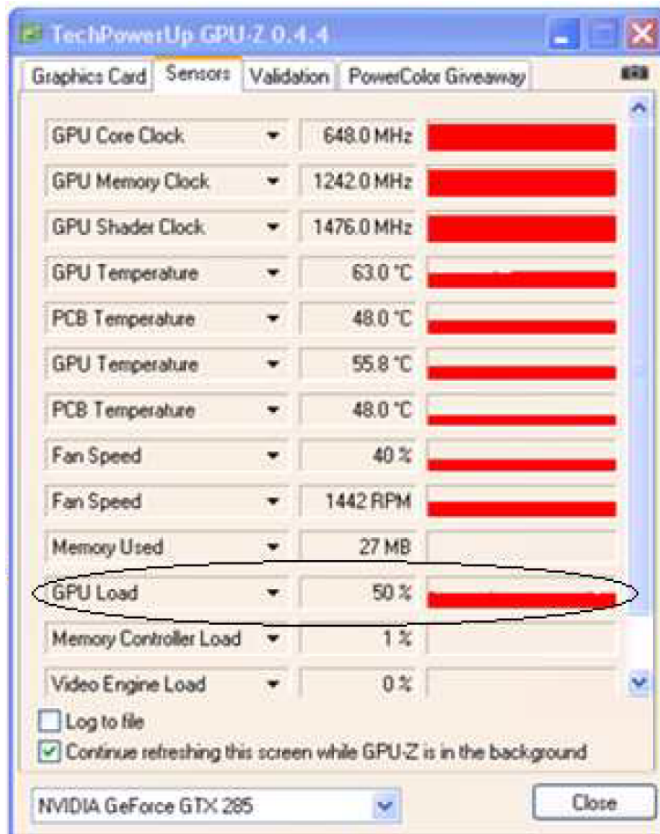


Figure 8. GPU load application

execution period for each test was calculated based on the values found in these files. Throughput measurements were made for loads ranging between 0% and 60% on the CPU and 0% to 75% on the GPU. The throughputs calculated under each load were then compared to the unloaded values.

5 Results

To determine the effects of GPU offloading for the applications without any additional system workload, the effective CPU time (t_{CPU}) of each GPU implementation was calculated by subtracting the measured GPU execution time (t_{GPU}) from the measured total execution time (t_{Total}): $t_{CPU} = t_{Total} - t_{GPU}$. To determine the percentage of CPU time saved by offloading computation to the GPU (t_{Saved}), this effective CPU time (t_{CPU}) was then compared to the time required for the application to compute solely on the CPU without any GPU offloading of computations (t_{NoGPU}).

$$t_{Saved} = \frac{t_{NoGPU} - t_{CPU}}{t_{NoGPU}} 100\% \quad (1)$$

Figs. 9 and 10 show the percentage of CPU time saved versus dataset size from offloading AES and Keccak, (respectively). A performance benefit from GPU offloading of these cryptographic applications was observed for datasets of 256 KB and larger. The best improvement was in AES-256, for which GPU offloading reduced CPU time by 60%. All implementations realized a time savings of at least 20% for datasets of 256 KB and larger, and the time savings increased with dataset size.

The remainder of this section describes performance results for the test applications running in a system with additional workloads. First the results of additional CPU loads are given. Next the effects of additional GPU loads are presented.

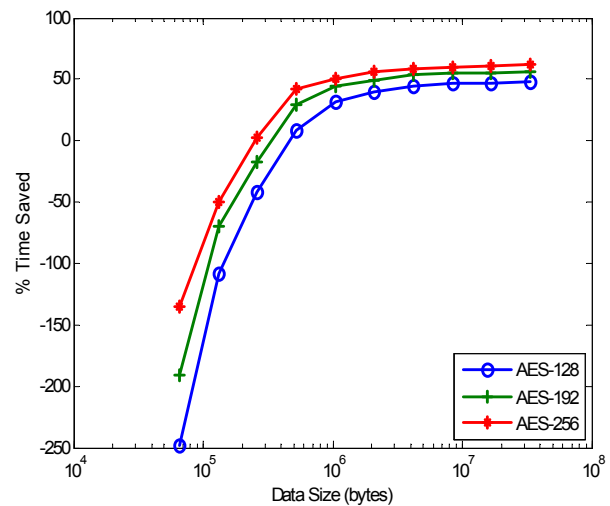


Figure 9. Offloading effects for AES

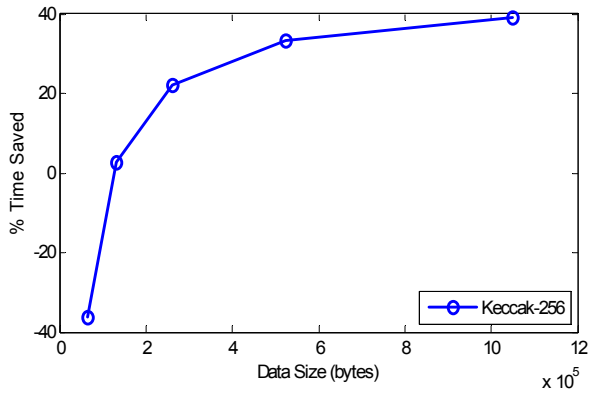


Figure 10. Offloading effects for Keccak

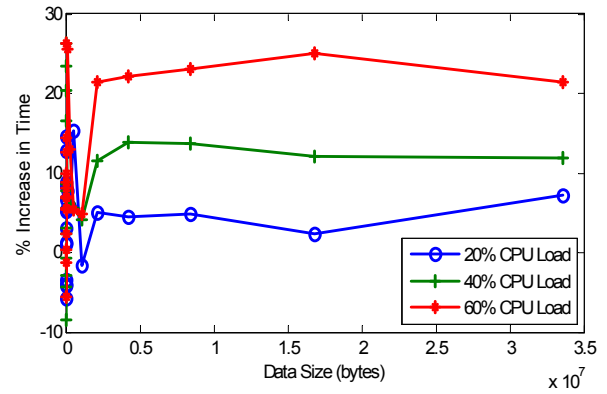


Figure 11. AES-256 CPU load effects

5.1 CPU load effects

Fig. 11 shows the effects of CPU load for AES-256. AES-128 and AES-192 performed similarly. CPU loading effects for Keccak-512 are illustrated in Fig. 12. For dataset sizes over 1 KB, the total computation time for Keccak is significantly longer than for AES.

For AES with its much lower total time, a CPU load may have a significant negative effect. With a 20% load, datasets larger than 1 MB experienced an average performance degradation of 4%. Average degradations of 12% and 22% are experienced with 40% and 60% loads, respectively, for datasets larger than 1 MB. The precise effects on datasets smaller than 1 MB are difficult to measure as they are generally encrypted quite fast and are prone to experience large percentage increases and decreases with small variations in performance. However, the effects on smaller datasets are generally minimal in terms of time. Thus, the effects of CPU loads are expected to increase as total time of the algorithm decreases.

In contrast, for the higher total time of Keccak, a CPU load has a negligible impact because the majority of the total time is from GPU processing. CPU loading effects for Keccak-512 are illustrated in Fig. 12. The bottom graph shows results for datasets larger than 128 KB, which are not distinguishable in the top graph. No degradation in execution time greater than 1.5% was measured for files larger than 32 KB. Again, the effects on datasets smaller than 32KB are difficult to measure but are minimal in terms of time.

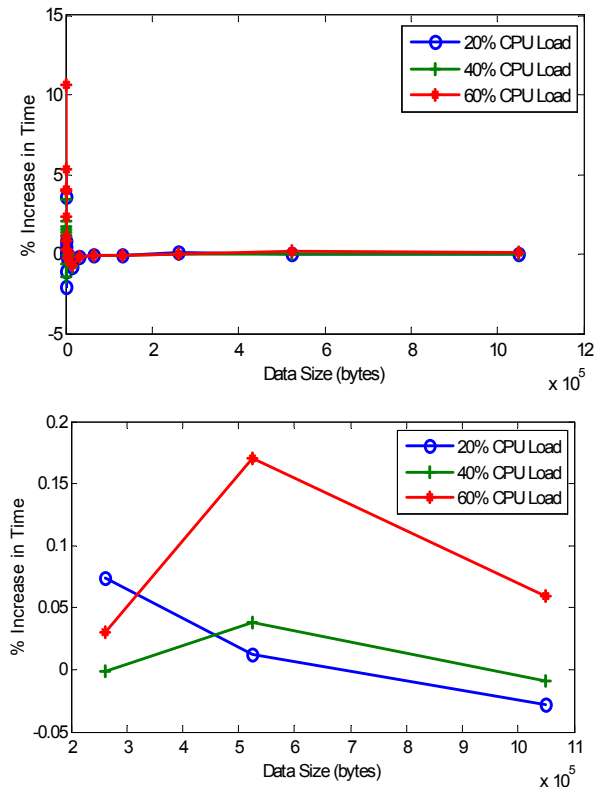


Figure 12. Keccak-512 CPU load effects

CPU and GPU. This additional data transfer requirement increases the load on the already slow PCI Express bus.

5.2 GPU load effects

Like the CPU load, the effects of introducing a GPU load are highly dependent on the execution time of the algorithm. However, the GPU load effects are quite different in nature from the CPU load effects. A GPU load consumes some portion of the resources available on the GPU and therefore makes those resources unavailable for a test application. Furthermore, a typical application producing the GPU load would also be expected to transfer data between the

The primary effect of GPU utilization is on total time, which is shown in Fig. 13 for AES-256. The corresponding graph for Keccak-512 is not included here since the wide variation in magnitude among the data points causes them to appear compressed along the axes when plotted on the same scale. Instead, the top graph in Fig. 14 plots dataset sizes smaller than 8 KB, and the bottom graph shows 8 KB and larger sizes. Since GPU utilization affects both the GPU and CPU, it produces a greater increase in total execution time than does a purely CPU load. The overall trend, however, is similar to that of CPU load effects.

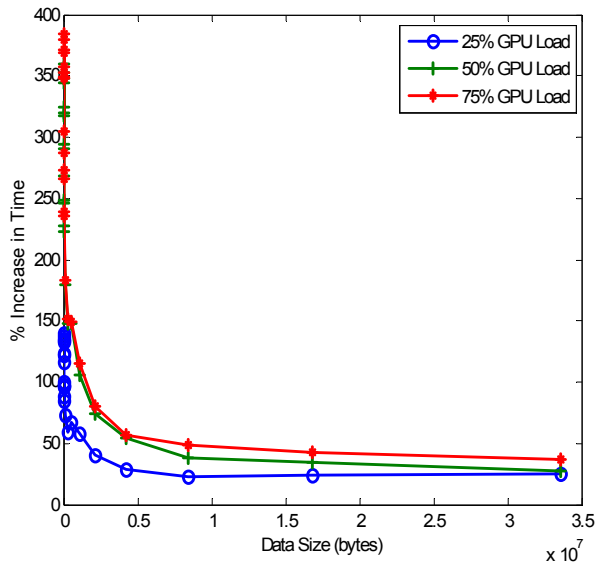


Figure 13. AES-256 GPU load effects on total time

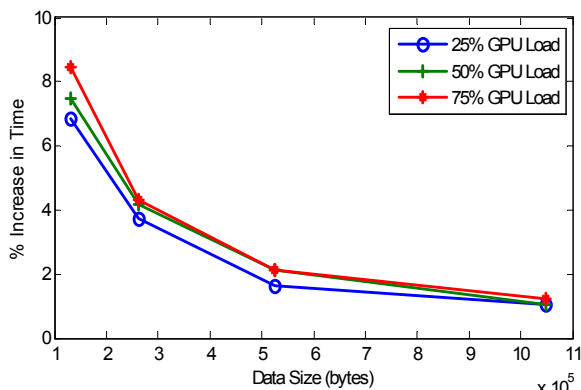
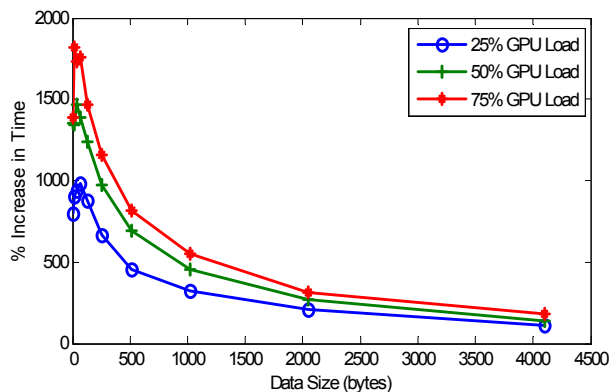


Figure 14. Keccak-512 GPU load effects on total time

As observed with CPU loading, the GPU loading results are highly dependent on total time of the algorithm. With slower algorithms such as Keccak-512, as the dataset size increases, the increase in time approaches 2%. These results likely indicate that the GPU resources were underutilized by the Keccak implementation and thus are available to support additional computation. On the other hand, for faster algorithms like AES, as the dataset size increases, the increase in time is much larger. In the case of AES-256, the increase

in time approaches 25%, 27%, and 36% for loads of 25%, 50%, and 75%, respectively. Smaller datasets are not capable of masking the effects of the load well and thus are more affected with greatly increased execution times up to 2000%.

The effects of GPU utilization on GPU time are less noticeable than those on total time, as seen in Fig. 15 for Keccak-512 on datasets smaller than 8 KB. (Again, a single graph of all data points is not included because the data points appear compressed along the axes. The general trend is similar to Fig. 14, and the percent increase in time is less than 0.2% for datasets of 8KB and larger.) Smaller datasets are affected more since they are hashed in very short times, which do not mask the overhead as well as larger datasets. They experience an increase in GPU time of 6–11% for Keccak-512. As the size of the dataset increases, the overhead is better masked, decreasing the percentage to 0–1% beyond 1 MB for AES-256 and beyond 32 KB for Keccak-512.

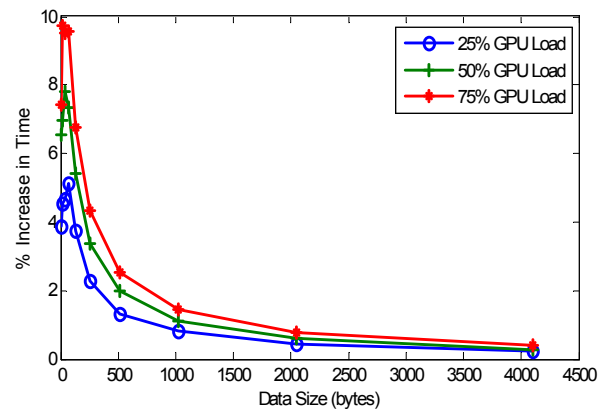


Figure 15. Keccak-512 GPU load effects on GPU time

6 Conclusions

To simulate a typical system environment, various CPU and GPU loads were introduced, and their effects on encryption and hashing performance using GPU co-processing were measured. CPU loads were found to have no effect on GPU time, but they did increase the total execution time. AES-256 experienced the largest increase by as much as 22% total execution time, but Keccak-512 saw minimal increases in total time because the CPU load effect was masked by the GPU execution time. GPU loads had similar effects on total time, although to a greater degree than CPU loads. The total time of AES-128 increased as much as 36%, whereas Keccak experienced minimal increases in total time.

For GPU processing, the effects of additional CPU loads are highly dependent on the total time of the algorithm. For such applications, the primary requirements of the CPU are for transferring data and initializing the kernel. Consequently, adding a CPU load has no effect on GPU processing time. However, total time can be adversely affected.

Similarly to CPU loads, the effects of introducing additional GPU loads are highly dependent on the total execution time of the algorithm; however, the effects are quite different in nature. A GPU load consumes some portion of the resources available on the GPU, which become unavailable for the GPU application. In addition to GPU computation, a typical GPU load also requires data transfer between the CPU and GPU. This extra data transfer requirement increases the load on the PCI Express bus, which increases total execution time.

As GPU processing for non-graphics applications becomes more common, such applications will certainly be deployed on platforms, such a general desktop computer, that have CPU and/or GPU loads in addition to the application. This investigation is a first step toward characterizing the effects of additional CPU and GPU loads on the performance of a GPU application. In this work, the effects of these additional types of loads have been considered independently of each other. Further investigation should evaluate both types of additional loads present simultaneously, as one would expect on a typical desktop system. Also, these effects have been evaluated only relative to the performance of two GPU encryption algorithms. Testing with other GPU applications is needed to see if they experience the same effects observed in this investigation. Also, performance effects on a variety of GPU applications with varying communication versus computation profiles need to be evaluated.

7 Acknowledgements

The authors wish to thank their colleagues at the Rochester Institute of Technology who contributed to this work: Dr. Andreas Savakis for equipment support from the Computer Engineering Real-time Vision and Image Processing Lab, and Dr. Muhammad Shaaban for helpful comments on drafts of this paper.

8 References

- [1] David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Burlington, MA: Morgan Kaufmann Publishers, 2010.
- [2] NVIDIA Corporation, "NVIDIA CUDA Programming Guide, Version 2.3.1," August 29, 2009, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.
- [3] NVIDIA Corporation, "CUDA Zone," [July 2010] <http://www.nvidia.com/cuda>.
- [4] NVIDIA Corporation, "NVIDIA CUDA C Programming Best Practices Guide, CUDA Toolkit 2.3," July 2009, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPractices_Guide_2.3.pdf.

[5] Max Bobrov, "Cryptographic Algorithm Acceleration Using CUDA Enabled GPUs in Typical System Configurations," master's thesis, Department of Computer Engineering, Rochester Institute of Technology, Rochester, NY, August 2010.

[6] National Institute of Standards and Technology, "Advanced Encryption Standard (FIPS-197)," 2001.

[7] Morris Dworkin, "Recommendation for Block Cipher Modes of Operation," NIST, Gaithersburg, MD, Special Publication 800-38A, 2001.

[8] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche, "Keccak Sponge Function Family Main Document, Version 2.1," June 19, 2010, <http://keccak.noekeon.org/Keccak-main-2.1.pdf>.

[9] National Institute of Standards and Technology, "Cryptographic Hash Algorithm Competition," December 13, 2010, <http://csrc.nist.gov/groups/ST/hash/sha-3/index.html>.

[10] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche, "Keccak Specifications, Version 2," September 10, 2009, <http://keccak.noekeon.org/Keccak-specifications-2.pdf>.

[11] Meltem Sönmez Turan, Ray Perlner, Lawrence E. Bassham, William Burr, Donghoon Chang, Shu-jeen Chang, Morris J. Dworkin, John M. Kelsey, Souradyuti Paul, and Rene Peralta, "Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithmic Competition," NIST, Gaithersburg, MD, Interagency Report 7764, February 2011.

[12] techPowerUp. "GPU-Z," July 2010, <http://www.techpowerup.com/gpuz/>.

Implementation and Evaluation of Program Development Middleware for Cell Broadband Engine Clusters

Toshiaki Kamata¹, Masahiro Yamada¹, Akihiro Shitara¹,
Yuri Nishikawa¹, Masato Yoshimi² and Hideharu Amano¹

¹Graduate School of Science and Technology, Keio University, 3-14-1 Hiyoshi Kouhoku-ku
Yokohama, Kanagawa 223-8522, Japan

²Faculty of Science and Engineering, Doshisha University, Tatara Miyakodani
Kyotanabe, Kyoto 610-0394, Japan
Email: cell@am.ics.keio.ac.jp

Abstract—Although PC clusters with multi-core accelerators have become popular, it is still difficult to write efficient parallel programs because two types of programming techniques are required: multi-thread programming and inter-node programming. The former requires special techniques and training dedicated to the accelerator, while the latter urges programmers to be skilled in using communication libraries such as mpich or OpenMPI. In order to reduce such programming cost, in this report, we propose a program development middleware which targets a PC cluster consisting of multiple nodes with Cell Broadband Engine (Cell/B.E.). This middleware supports inter-node and inter-core thread control, so it lets developers to focus on tuning a program to elicit computational power of each core in Cell/B.E. processors. As a result of evaluating middleware by executing two types of benchmark programs, it could reduce 40% of code quantity compared to OpenMPI implementation, and provided approximately the same execution performance.

Keywords: Cell Broadband Engine, Virtualization, Parallel Computing

1. Introduction

PC clusters with multi-core accelerators such as Cell Broadband Engine (Cell/B.E.), Graphic Processing Unit (GPU), ClearSpeed, and Field Programmable Gate Array (FPGA) have become popular especially for high performance scientific computing[1][2][3][4]. Especially, a Cell/B.E. cluster consisting multiple PlayStation3 nodes is considered as flexible and cost-effective computing environment because the processor MIMD-based, and unit price of PlayStation3 is affordable despite its high computation power. Examples of PlayStation3 clusters can be found in [5] and [6].

However, writing efficient program on Cell/B.E. cluster is yet a difficult task. In the first place, stand-alone Cell/B.E., programming requires acquirement of dedicated programming language and tuning techniques. In concrete, (1) programmers have to write two separate programs that run on controlling core and computation cores by using

libspe2 and pthread libraries, and (2) data transfers between memories and multiple cores need to be explicitly specified using DMA transfer instructions. In addition to this, for using multiple Cell/B.E. processors in a cluster environment, inter-node communication should be described in order to control multiple nodes using communication libraries such as OpenMPI or mpich[7]. This difficulty of the programming forms the main reason why Cell/B.E. clusters are not popularly used by end-users, in spite of Cell/B.E.'s distinctive potential in terms of flexibility and cost-performance.

In this paper, we propose a middleware which mitigates such difficulty on program development of Cell/B.E. clusters. Using this middleware, a programmer can focus only on SPE programming and tuning without writing PPE and inter-node communication program codes. The programming environment with the proposed middleware is now available on a PC cluster consisting of an Intel Xeon node and several SONY BCU-100 nodes. The evaluation results using two applications: Monte-carlo method and matrix product appeared that the overhead of the middleware is acceptable considering its benefit in programming.

The rest of this report is organized as follows: Section 2 describes Cell/B.E. and parallel processing using multiple Cell/B.E. in cluster environment. Section 3 is for related work. Section 4 shows the design, and Section 5 shows the implementation of this middleware. Section 6 shows the evaluation. Finally, we state the conclusion and future work in Section 7.

2. Cell Broadband Engine

Cell Broadband Engine (Cell/B.E.) is a multi-core processor jointly developed by SONY, Toshiba and IBM known as STI as core of PlayStation3. Figure 1 shows the structure of Cell/B.E..

Cell/B.E. and other processors based on Cell Broadband Engine Architecture (CBEA) are classified in a heterogeneous processor with a general purpose 64-bit processor based on PowerPC called PowerPC Processor Element (PPE), and eight SIMD processors called Synergistic Processor Element (SPE). Each processor is connected by a ring

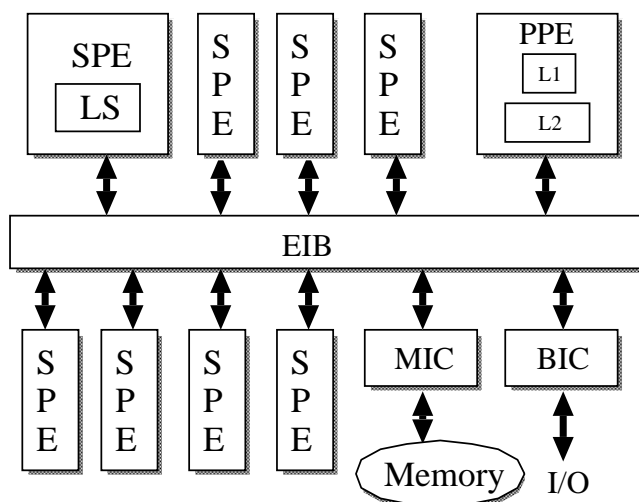


Fig. 1: The architecture of Cell/B.E.

based interconnect called Element Interconnect Bus (EIB). Eight SPEs run in parallel, (six in case of a Cell/B.E. used in PlayStation3), and total performance is 204.8GFlops for single precision floating point calculations[8].

PPE is a general purpose processor which runs the operating system, and also controls SPEs, main memory and other external devices. It consists of a PowerPC Processing Unit (PPU) connected to 32KB L1 cache and 512KB L2 cache. PPU has VMX, a 128-bit SIMD unit, which is based on PowerPC instruction set architecture.

SPE is a 128-bit SIMD processor consisting of a Synergistic Processor Unit (SPU), Local Store (LS) and Memory Flow Controller (MFC). It is suitable for multimedia processing such as image processing, and MPEG stream encoding/decoding. A programmer can control eight SPEs from PPE using high level programming language such as C/C++ with `libspe2` libraries. The 128-bit register can store four single precision floating point numbers. The rounding of SPE's floating point number does not follow IEEE754 standard. It has a 256KB exclusive memory called Local Store (LS), and accessed by 128-bit per cycle.

EIB has a ring structure consisting of four buses each of which can transfer 16-Byte in a cycle. Thus, the total performance of EIB is 96-Byte/cycle. It is used for data transfer among PPE, SPEs, Memory Interface Controller (MIC) and Bus Interface Controller (BIC). These transactions occur simultaneously within the same ring.

When a common program without any consideration runs on Cell/B.E., only PPE will be used. In order to use SPEs, the programmer has to write two different programs running on PPE and SPE, and declares "context" (object to control SPE in software level) in PPE source code to control each SPE. Generally, threads are created according to the number of utilized SPEs, and they are controlled by the PPE. Since

SPE can only access data stored in its LS, the target data to be processed must be transferred from the main memory using DMA transfer. The DMA transfer has some limitation on memory alignment and data size, thus, the program must take care of such limitations[9].

3. Related Work

In this section, we introduce related works which aims efficient use of computational resources of multi-core processor in cluster environments.

As examples of Cell/B.E. cluster programming environments or frameworks, there is a proposal of programming framework by Kunzman which provides programmers a technical guideline to efficiently load-balance tasks among multiple Cell/B.E. nodes[10]. They suggest automatic off-loading methodology of PPE's tasks to SPEs, and management technique of the tasks by adopting job queues. They evaluated performance by using homogeneous PlayStation3 cluster. Also there is a proposal of a communication API by Pakin called Cell Messaging Layer (CML), which is implemented by MPI[11]. The performance of the CML is analyzed with homogeneous cluster of multiple IBM's BladeCenter QS21, which is a blade server that equips two Cell/B.E.s Yamada proposed Thread Virtualization Environment (TVE), a middleware which shows an image to programmers as if SPEs in multiple Cell/B.E.s connected a network are integrated on a single processor, and one PPE on a host machine can use all the SPEs[12]. By using TVE, programmers can write a parallel distributed program without use of communication libraries.

As examples of multi-core cluster middleware, Ninf project is a representative programming middleware for efficient use of computational resources in grid environments[13]. It can offload heavy tasks to remote cluster environments with larger computational capacity. It is suitable for developing flexible and fault-tolerant grid system whose node size may change frequently. This can not be realized by MPI programming which requires deterministic assignment of nodes that the program run on.

Also, XcalableMP released in November 2010, can automatically generate parallel program codes which can be applied in cluster environment by inserting `#pragma` directives in to non-parallelized program codes. XcalableMP provides Java-based libraries for general-purpose processor such as Intel X86, which can hide MPI communications.

Compared to the related works presented above, the middleware that we propose in this paper has following characteristics:

- It assumes a cluster environment with x86 servers and multiple Cell/B.E. nodes with different number of SPEs,
- provides libraries for automatic inter-node communication tuning and intra-node control, and

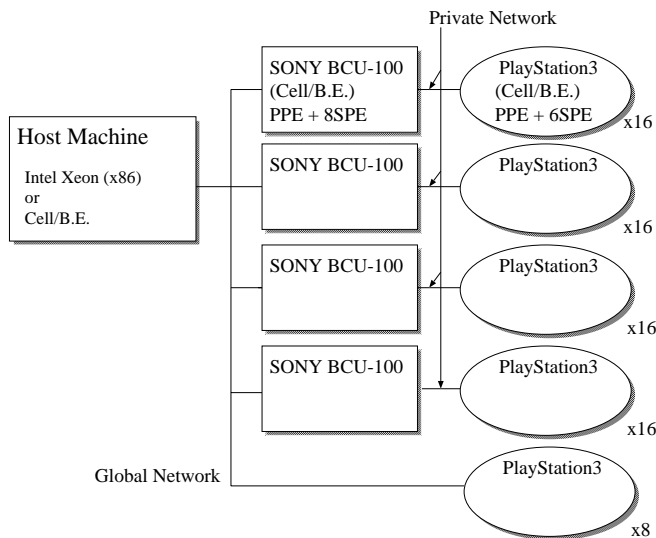


Fig. 2: Example of PC cluster environment with multiple Cell/B.E.

- provides high flexibility to the modification of the system structure.

In the next section, we describe the design of this middleware.

4. Design of Program Development Middleware

In this section, we explain the design of middleware for PC cluster environment with multiple Cell/B.E. processors.

Generally, in order to run a parallel program in such an environment requires communication program codes between host and client machines, and program codes for a PPE to control its subordinate SPEs by using libspe2 and pthread libraries. This urges a programmer to learn two types of programming techniques, and also results in large code quantity. In order to mitigate such a programming burden, we propose a Virtual SPE programming environment.

This mechanism allows programmers to focus on tuning SPE codes, and brings out an SPE's computing power by releasing them to describe communication part by using socket connection with thread control functions.

4.1 Target Environment

We assume an environment shown in Figure 2.

- Multiple client Cell/B.E. nodes are connected by a network such as Ethernet.
- Multiple client Cell/B.E. nodes with different number of SPEs are connected.
- A host machine (Intel x86 processor or Cell/B.E.) offloads its tasks to client Cell/B.E. nodes.

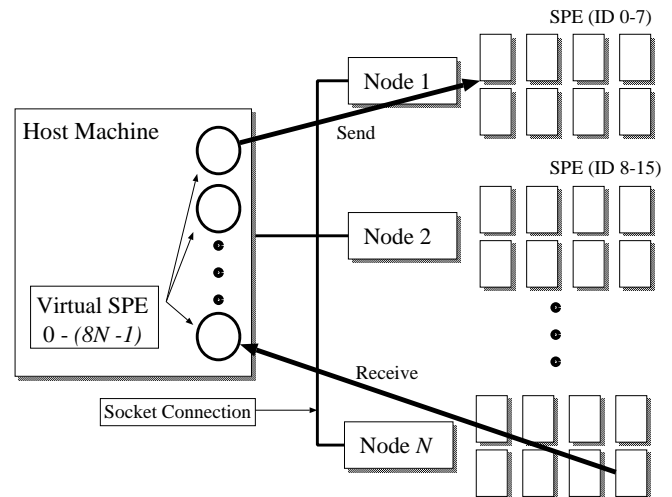


Fig. 3: Outline of Virtual SPE environment

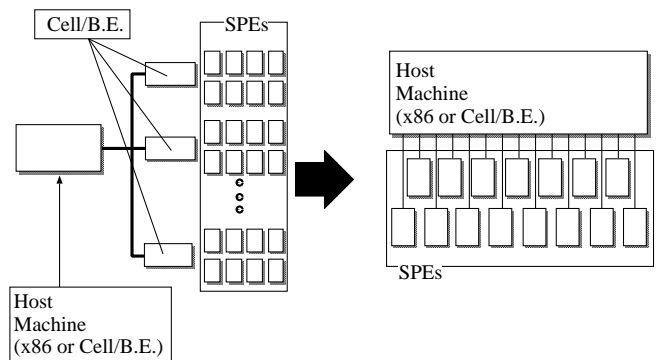


Fig. 4: Virtualization by middleware

Here, PlayStation3 (PS3) and SONY BCU-100 are examples of machine with Cell/B.E.. PS3 have 7 SPEs, and BCU-100 have 8 SPEs. Each Cell/B.E. are connected with host machine.

4.2 Required Feature

Figure 3 shows the outline of Virtual SPE environment. When a programmer issues operation to a Virtual SPE on the host machine, physical SPE of each node corresponding to the Virtual SPE executes it. Although programmers must know the number of physical SPEs in the system and declare it, they can be treated as if they were connected with the host directly as shown in Figure 4. Thus, they are called "Virtual SPE". Required feature of this middleware are following:

- Communication between host and node machines.
- Middleware can send or receive arbitrarily sized data between host machine and specified SPE.
- Connection and data transfer between other nodes without host machine.

Table 1: List of communication functions

Function Name	Feature
API_Initialize	Connects to a server program. This function is called automatically at the beginning of a program.
API_Finalize	Closes all connections. This function is called automatically at the end of a program.
Barrier_All	Waits until all SPE's executions are terminated.

Generally, in order to use Cell/B.E. requires two program files running PPE and SPE, and control part of SPE context and pthread function is the most. In contrast, using this middleware, PPE program is eliminated and programmer can focus optimizing of SPE program.

In next section, we explain about implementation of this middleware.

5. Implementation

Here, the implementation of the following mechanisms supported by the middleware is shown.

- 1) Server program for transferring data between a host and client machines, and between the PPE and SPEs.
- 2) A "Virtual SPE" in order to use SPEs beyond the network.

We also show an example program code which uses the Virtual SPE. Table 1 shows the list of communication functions between a host and client machines. Virtual SPEs can be used only by including an original header file. Then, functions *API_Initialize* and *API_Finalize* are called automatically at the beginning and the end of the program execution.

5.1 Server program on PPE

In general, data cannot be directly transferred between a host machine and SPEs. Thus, we implemented a server program which supports data transfer among a host and node machines. It runs on a PPE in each client machine, and receives commands from the Virtual SPE in the host machine in order to control the corresponding physical SPEs. A server program has the following functions:

- It receives data from host machine using socket connection,
- manages threads according to the number of SPEs, and
- sends data to a specified SPE by the DMA transfer when it receives data from a host machine. The transfers are repeated when data size exceeds 16 KB, a maximum data size that can be sent at once.
- Then it sends data to a host machine from data buffers in subordinate SPEs.

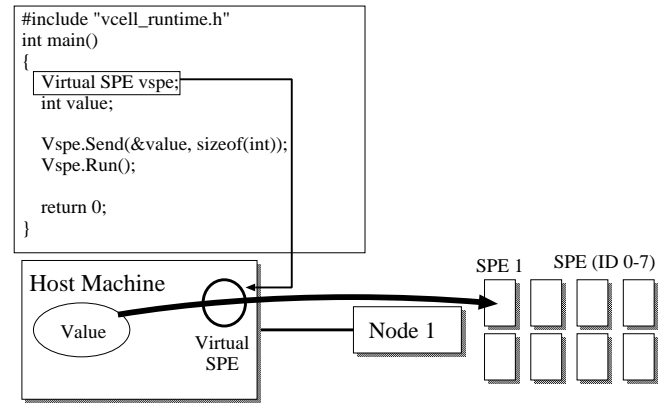


Fig. 5: Example using Virtual SPE environment

Table 2: List of Virtual SPE functions

Function Name	Feature
Vspe.Send	Send data to target SPE
Vspe.Recv	Recv data to target SPE
Vspe.Run	Start SPE execution
Vspe.Wait	Wait until the end of execution

As server program provides above functions, a programmer can only focus on the distribution and tuning of SPE programs.

Double- or multi-buffering is a popular tuning technique. An SPE prepares multiple buffers to hold blocks of data, and initiates DMA transfer for data used in the next step (e.g. next loop), while processing computation. In other words, this technique can hide memory latency by overlapping with computation. In our middleware, it can be applied to SPE program unless multiple buffer sizes exceed that of LS.

5.2 Virtual SPE

Here, the outline of a Virtual SPE shown in Figure 5 is introduced. It is an C++ class object, and corresponds to a physical SPE one by one.

Table 2 shows functions to control Virtual SPEs. A programmer can control data transfer or initialize of program execution for remote SPEs by using Virtual SPE functions listed above.

Details of functions are as follows:

- Vspe.{Send, Recv}

These functions support sending or receiving data between a host machine and target SPEs. Arguments of this function are memory address of data, data size, and target SPE number. Data transfer between clients and host machines, and DMA transfer between PPE and SPEs are initialized by the PPE server program automatically. According to Cell/B.E.'s specification, the DMA transfer can send 16KB of data block in maximum. When data size exceeds this, the PPE server

Table 3: The environment of evaluation

	Host machine	Node machine
Hardware	Intel Xeon	SONY BCU-100
CPU	Intel Xeon 2GHz	Cell/B.E. 3.2GHz
Memory	2GB	1GB
OS	CentOS 5.4	Yellow Dog Linux 6.0
Compiler	g++-4.1.2	{ppu, spu}-g++ 4.1.1

divides them into multiple blocks and repetitively transfers data

- **Vspe.Run** This function starts the execution of target SPE's. It only issues an execution (run) command to a specified SPE, and does not wait for acknowledgment of its initialize nor termination. If a programmer wants to synchronize with other SPE, they need to use "Vspe.Wait" function.
- **Vspe.Wait** This function is used when a programmer wishes to wait until specified SPE's execution is done.

Programmers first must declare several Virtual SPE objects according to the number of SPE to be used, and control them by using the above functions. After Virtual SPEs are declared, ID numbers are given to each node and SPE automatically by the middleware.

6. Evaluation

In this section, we evaluate the availability of Virtual SPE environment with the following two applications:

- Calculation of circular constant by using Monte-Carlo integration
- Matrix-Matrix product

Performance is evaluated with the environment shown in Table 3.

Virtual SPE environment lets programmers to use as many SPEs as connected to the same network segment with a host machine. The total number of SPE available is equal to the number of BCU-100 \times 8 plus the number of PlayStation3 \times 7. Host machine in this evaluation equips Intel Xeon processor, and four SONY BCU-100 servers as client nodes.

First, we confirm acceleration effect by parallel processing in this environment by using Monte-Carlo integration program which requires small amount of data transfer. As this is an embarrassingly parallel algorithm, computational load of each thread or process is uniform, and N , the total count of plots, is simply divided by the number of SPEs. Thus, linear speedup according to the number of used SPEs is expected.

Figure 6 shows the relationships between execution time and number of SPEs for the case of using the middleware and OpenMPI implementation. The figure shows that the middleware can achieve approximately the same parallel effect with OpenMPI implementation, and performance would further approach as plot count increases. The overhead of

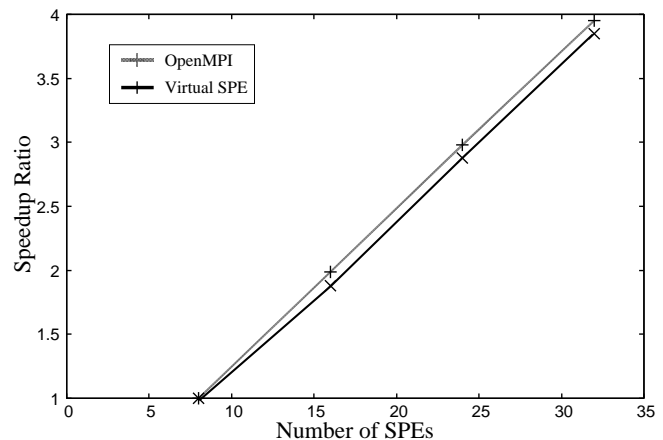


Fig. 6: Relation between execution time and number of SPEs (plotting count $N = 10^{10}$)

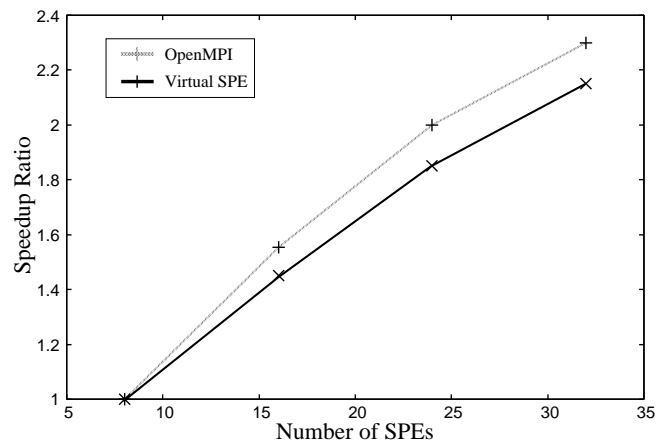


Fig. 7: Relation between execution time and number of SPEs (Dimension size of Matrix $N = 4096$)

the middleware is caused when connections to all nodes are established, but its impact becomes smaller as the algorithm becomes more computation bound.

Second, we evaluate performance using matrix-matrix product benchmark. Figure 7 and Figure 8 show the relationship between execution time and dimension size of the matrix. As shown in Figure 8, performance of our middleware is about 80% of OpenMPI implementation. It is noted that the performance is degraded when the matrix size is large, due to increase of communication overhead, which derives from both socket communication and frequent DMA transfers.

Third, we evaluate programmability of this middleware in terms of code quantity. Table 4 shows the code size of computation portion of the program in case of using OpenMPI, libspe2 and our middleware. The table indicates that our middleware can reduce approximately 40% of code quantity.

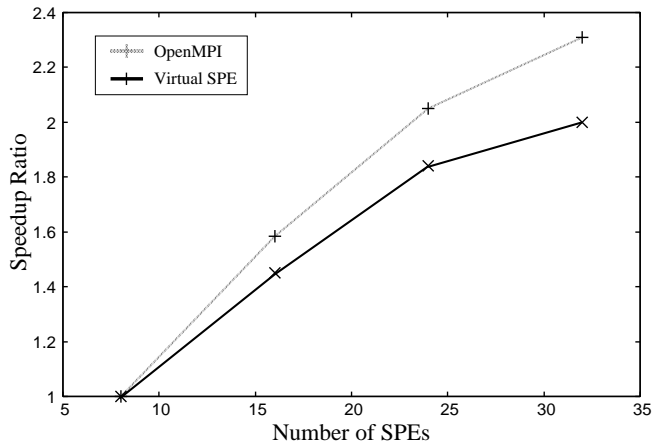


Fig. 8: Relation between execution time and number of SPEs (Dimension size of Matrix $N = 8192$)

Table 4: Program steps of benchmark application

	Virtual SPE	OpenMPI + libspe2
Monte-Carlo	150 (Node + SPE)	250 (PPE + SPE)
Matrix-Product	200 (Node + SPE)	340 (PPE + SPE)

The evaluation results of Monte-Carlo integration, Matrix-matrix product and code quantity suggest that the overhead of the middleware is acceptable considering its benefit in programming.

We didn't evaluate node-to-node communication. It can apply to some applications which required data transfer without host machine. In addition, The future works are following:

- 1) Evaluation with 10 Gigabit Ethernet and Infiniband network environment
- 2) Examination of another selection method of each SPE.
- 3) Visualize utilization of each SPE.

To evaluate with network environment listed above will cancel the overhead of data transfer, and It can achieve more closer performance compared with OpenMPI.

In this implementation, Virtual SPE correspond to physical SPE one-to-one. the alternative method of this is using Virtual SPE depends on distance of physical network. We considered this mechanism in evaluation, however, difference between communication of intra-node (same Cell/B.E.) and inter-node makes complexity. Finally, In this middleware, there is no method to observe activity of each SPE. therefore, programmer doesn't know each SPE's load.

7. Conclusion

In this paper, we proposed and evaluated Virtual SPE environment, and compared with traditional OpenMPI implementation.

The evaluation results reveal following:

- Highly parallel application such as Monte-carlo method or Matrix product, the Virtual SPE environment achieved same performance of OpenMPI execution in maximum
- Middleware can reduce 40% of these program steps

Here, the evaluation was done using a cluster with Intel Xeon and SONY BCU-100 connected by Ethernet. Thus, the network easily forms a bottleneck and the type of applications which can be efficiently executed are limited. Evaluation with various types of application on the other platform is our future work.

References

- [1] K.J.Barker, K.Davis, A.Hoisie, D.J.Kerbyson, M.Lang, S.Pakin, and J.C.Sancho, "Entering the Petaflop Era: The Architecture and Performance of Roadrunner," in *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008, pp. 1 – 11.
- [2] S. Matsuoka, "The TSUBAME Cluster Experience a Year Later, and onto Petascale TSUBAME 2.0," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, F. Cappello, T. Herault, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2007, vol. 4757, pp. 8–9, 10.1007/978-3-540-75416-9_5.
- [3] K. Tsoi and W.Luk, "Axel: A Heterogeneous Cluster with FPGAs and GPUs," in *In Proceedings of International Symposium of Field Programmable Gate Array (FPGA)*, 2010.
- [4] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton, "Programming the Linpack benchmark for the IBM PowerXCell 8i processor," *Scientific Programming*, vol. 17, no. 1-2, pp. 43–57, 2009.
- [5] J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra, "The playstation 3 for high performance scientific computing," 2008.
- [6] A. Buttari, J. Dongarra, and J. Kurzak, "Limitations of the PlayStation3 for high performance cluster computing," Tech. Rep., 2007.
- [7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance portable implementation of the mpi message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sept. 1996.
- [8] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its first implementation: A performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559 –572, 2007.
- [9] I. Corp, "'Cell/B.E. Programming Handbook Version 1.1,'" <http://www.ibm.com/developerworks/power/cell/>.
- [10] D. M. Kunzman and L. V. Kale, "Towards a framework for abstracting accelerators in parallel applications: Experience with cell," in *SC'09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, 2009, pp. 1– 2.
- [11] S. Pakin, "Receiver-initiated message passing over rdma networks," in *In Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, 2008, pp. 1– 2.
- [12] M. Yamada, Y. Nishikawa, M. Yoshimi, and H. Amano, "A Proposal of Thread Virtualization Environment for Cell Broadband Engine," in *In Proceedings of Parallel and Distributed Computing and Systems (PDCS)*, no. 724-027, 2010.
- [13] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, "Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing," *Journal of Grid Computing*, vol. 1, no. 1, pp. 41–51, 2003.

Performance Analysis and Evaluation of LANL's PaScalBB I/O nodes using Quad-Data-Rate Infiniband and Multiple 10-Gigabit Ethernet Bonding

Hsing-bugn Chen, Alfred Torrez, Parks Fields
 HPC-5, Los Alamos National Lab
 Los Alamos, New Mexico 87111, USA
 {hbchen, atorrez, parks}@lanl.gov

Juan C. Franco, Daniel Illescas, Rocio Perez-Medina,
 Jharrod LaFon, Ben Haynes, John Herrera
 INST-OFF, HPC Summer School
 Los Alamos National Lab

Abstract - In the LANL's PaScalBB network I/O nodes carry data traffic between backend compute nodes and global scratch based file systems. An I/O node is normally equipped with one Infiniband Nic for backend traffic and one or more 10-Gigabit Ethernet Nics for parallel file system data traffic. With the growing deployment of multiple, multi-core processors in server and storage systems, overall platform efficiency and CPU and memory utilization depends increasingly on interconnect bandwidth and latency. PCI-Express (PCIe) generation 2.0 has recently become available and has doubled the transfer rates available. This additional I/O bandwidth balances the system and makes higher data rates for external interconnects such as Infiniband feasible. As a result, Infiniband Quad-Data Rate (QDR) mode has become available on the Infiniband Host Channel Adapter (HCA) with a 40 Gb/sec signaling rate. Combining HCA QDR data rates with multiple 10-Gigabit Ethernet links and using it in an IO node has created the potential to solve some of the I/O traffic bottlenecks that currently exist. We have setup a small-scale PaScalBB testbed and conduct a sequence of I/O node performance tests. The goal of this I/O node performance testing is to figure out an enhanced network configuration that we can apply to the LANL's Cielo machine and future LANL HPC machines using PaScalBB architecture.

Keywords- *Server I/O networking, High Performance Networking, Infiniband, 10 Gigabit Ethernet, Link aggregation, Load balancing*

1. INTRODUCTION

Commercial off the shelf based cluster computing Systems have delivered reasonable performance to technical and commercial areas for years. High speed computing, global storage, and networking (IPC and I/O) are the three most critical elements to build a large scale HPC cluster system. Without these three elements being well balanced, we cannot fully utilize a HPC cluster. High data bandwidth I/O networking provides a data super-highway to meet the needs of constantly increasing computation power and storage capacity.

LANL's PaScalBB server I/O architecture is designed to support data-intensive scientific applications running on very large-scale clusters. The main goal of PaScalBB is to provide high performance, efficient, reliable, parallel, and scalable I/O capabilities for data-intensive scientific applications running

on very large-scale clusters. Data-intensive scientific simulation-based analysis normally requires efficient transfer of a huge volume of complex data among simulation, visualization, and data manipulation functions. To date PaScalBB has been implemented on most of HPC production machines at LANL; Roadrunner (1st Petaflops machine), RedTail, LOBO, Turing, TLCC, etc.

I/O nodes are used in the LANL's PaScalBB network to carry data traffic between backend compute nodes and global scratch based file systems. An I/O node is normally equipped with one Infiniband NIC for backend IPC traffic and one or more 10-Gigabit Ethernet NICs for parallel file system data traffic. With the growing deployment of multiple, multi-core processors in server and storage systems, overall platform efficiency and CPU and memory utilization depends increasingly on interconnect bandwidth and latency. PCI-Express (PCIe) generation 2.0 has recently become available and has doubled the transfer rates available. This additional I/O bandwidth balances the system and makes higher data rates for external interconnects such as Infiniband feasible. As a result, Infiniband Quad-Data Rate (QDR) mode has become available on the Infiniband Host Channel Adapter (HCA) with a 40 Gb/sec signaling rate. Combining HCA QDR rates with multiple 10-Gigabit IPC Ethernet links has the potential to solve some of the I/O traffic bottlenecks that currently exist. We have setup a small-scale PaScalBB test bed and conduct a sequence of I/O node performance tests. The goal of this I/O node performance testing is to figure out an enhanced network configuration that we can apply to the LANL's Cielo machine and future LANL HPC machines using PaScalBB architecture.

The rest of this paper is organized as follows. In section two we describe LANL's PaScalBB server I/O infrastructure. Section three introduces Infiniband/QDR and 10Gigabit Ethernet technologies. We then illustrate our experimental setup and discuss testing results and performance data in section four. Finally, we present our conclusion and future works in section five.

2. PASCALBB SERVER I/O BACKBONE ARCHITECTURE

LANL's PaScalBB [10] adopts several hardware and software components to provide a unique and scalable server

I/O networking architecture. Figure-1 illustrates the system components used in PaScalBB.

2.1 Hardware Components used in PaScalBB

2.1.1 Level-1 High Speed Interconnection Network

The Level-1 interconnect uses (a) high speed interconnect systems such as Quadrics, Myrinet, or Infiniband for fulfilling requirements of low latency, high speed, high bandwidth cluster IPC communication and (b) aggregating I/O-Aware multi-Path routes for load-balancing and failover.

2.1.2 Level-2 IP based Interconnection Network

The Level-2 interconnect uses multiple Gigabit Ethernet switches/routers with layer-3 network routing support to provide latency-tolerant I/O communication and global IP based storage systems. Without using the "Federated network" solution, we can linearly expand the Level-2 IP based network by employing a global host domain multicasting feature in metadata servers of a global file system. With this support we can maintain a "single name space" global storage system and provide a linear cost growing path for I/O networking.

2.1.3 Compute node

A Compute node is equipped with at least one high-speed interface card connected to a high-speed interconnect fabric in Level-1. The node is setup with Linux multi-path equalized routing to multiple available I/O nodes for load balancing and failover (high availability). A Compute node is used for computing only and is not involved with any routing activities.

2.1.4 I/O node

I/O node: An I/O routing node has two network interfaces. One high-speed interface card is connected to the Level-1 network for communication with Compute nodes. One or more Gigabit Ethernet interface cards (bondable) are connected to the Level-2 linear scaling Gigabit switches. I/O nodes serve as the routing gateways between Level-1 and Level-2 network. Every I/O has the same networking capability.

2.2 System Software Components used in PaScalBB

2.2.1 Equal Cost Multi-path routing for load balancing

Multi-path routing is used to provide balanced outbound traffic to the multiple I/O gateways. It also supports failover and dead-gateway detection capability for choosing good routes from active I/O gateways. Linux Multi-Path routing is a destination address-based load-balancing algorithm. Multi-path routing should improve system performance through load balancing and reduce end-to-end delay. Multi-path routing overcomes the capacity constraint of "single-path routing" and routes through less congested paths.

Each Compute node is setup with N-ways multi-path routes thru "N" I/O nodes. Multi-path routing also balances the bandwidth gap between the Level-1 and the Level-2 interconnects. We use the Equal Cost Multi-path (ECMP) routing strategy on compute nodes so compute nodes can evenly distribute traffic workloads on all I/O nodes.

With this bi-directional multi-path routing, we can sustain parallel data paths for both write (outbound) and read (inbound) data transfer. This is especially useful when applied to concurrent socket I/O sessions on IP based storage systems. PaScalBB can evenly allocate socket I/O sessions to routing available I/O routing nodes.

I/O nodes are used heavily in the LANL's PaScalBB network to carry data traffic between backend compute nodes and global scratch based file systems. An I/O node is normally equipped with one Infiniband NIC for backend IPC traffic and one or more 10-Gigabit Ethernet NICs for parallel file system data traffic [6][7][8].

3. INFINIBAND AND 10 GIGABIT ETHERNET

Infiniband [3] is a standard switched fabric communication link used in high performance computing and enterprise data centers. The InfiniBand Architecture (IBA) is designed to provide high bandwidth, low-latency computing; the scalability to support thousands of nodes and multiple processor cores per server; and efficient utilization of compute processing resources. The TOP-500 list published in November 2010 shows that more than 42% of the computing systems use Infiniband as their primary high-speed interconnecting network. The growth rate of Infiniband in the TOP-500 systems is about 30%. This is an indication of a strong momentum in adoption of Infiniband technology in HPC and Enterprise communities.

Ethernet has long been the dominant LAN technology. Now the availability of 10-Gigabit Ethernet has enabled new applications in the data center and IP based storage systems. Because 10-Gigabit Ethernet is based on the core Ethernet technology, it takes advantage of the wealth of improvement that has been developed over the years and simplifies the migration to this higher-speed technology.

With the growing deployment of multiple, multi-core processors in server and storage systems, overall platform efficiency and CPU and memory utilization depends increasingly on interconnect bandwidth and latency. PCI-Express (PCIe) generation 2.0 has recently become available and has doubled the transfer rates available. This additional I/O bandwidth balances the system and makes higher data rates for external interconnects such as Infiniband feasible. As a result, Infiniband Quad-Data Rate (QDR) mode has become available on the Infiniband Host Channel Adapter (HCA) with a 40 Gb/sec signaling rate. Combining Infiniband HCA QDR data rates with multiple 10-Gigabit Ethernet links and using it in IO node nodes has created the potential to solve some of the I/O traffic bottlenecks that currently exist in HPC machines.

4. EXPERIMENTAL TESTING SETUP AND PERFORMANCE EVALUATION

We setup a small-scale PaScalBB test bed and conduct a sequence of I/O node performance tests.

4.1 Testing setup and configuration

Hardware equipment includes

- (a) Twelve Linux server machine—Intel Nehalem 5600 DualQuad-core with 16GB DDR3 memory: seven Compute nodes with one Mellanox ConnectX Infiniband QDR on each compute node, one I/O node with Mellanox ConnectX Infiniband QDR [10] and multiple Mellanox ConnectX 10-Gigabit Ethernet Nics, and four data nodes with one 10-Gigabit Ethernet connection on each node,
- (b) One Mellanox 36-port Infiniband QDR switch, and
- (c) One Arista 24-port 10-Gigabit Ethernet Switch [11].

Software components include

- (a) Fedora 12/Linux64-bit OS,
- (b) OFED (OpenFabrics Enterprise Distribution) [9] Infiniband/10Gigabit Ethernet system software,
- (c) Linux Ethernet bonding driver, and
- (d) netperf [12] - a network performance benchmark software.

4.2 Performance testing and evaluation

4.2.1 Infiniband SDR/DDR/QDR performance testing

Figure-2 shows the one-way communications from IB/SDR(single data rate), IB/DDR(double data rate) and IB/QDR(quad data rate). This figure illustrates the improvement of 75% of bi-directional bandwidth when moving from DDR to QDR. Figure-3 shows the latency testing results from IB/SDR, IB/DDR, and IB/QDR. This result demonstrates the advantage of using QDR in terms of lower latency. Figure-4 shows that MPI I/O testing using various message packet sizes from 1MB to 200MB. This result shows that IB/QDR can persistently provide consistent bandwidth when various message sizes are applied in MPI applications. Figure-5 shows the results of (a) QDR/UC (unreliable connection) one way communication bandwidth, (b) QDR/RC (reliable connection) one way communication bandwidth, and (c) QDR/SRQ(shared receiving queue) bi-direction communication bandwidth. We can see that IB/QDR can reach a peak of 5600MB+/sec bi-directional bandwidth from multiple streams of netperf testing.

4.2.2 10-Gigabit Ethernet performance testing

Figure-6 shows the performance results for back-to-back connection using one single 10-Gigabit Ethernet link between two server nodes. We can reach 95% bandwidth of a physical 10-Gigabit link. Figure-7 shows the performance result from triple 10-Gigabit Ethernet bonding back-to-back connection. This figure illustrates that we can reach a peak 2300MB/sec bandwidth from three-10GiGE link bonding. Figure-8 shows the performance result from quad 10-Gigabit Ethernet bonding back-to-back connection. It only improve 5% -10% bandwidth compared it with the three-10-Gigabit Ethernet bonding. It may be due to the Ethernet chip-set processing capability or the Linux TCP/IP software stack.

4.2.3 I/O node performance testing and justification

Figure-9 shows the results of using four compute nodes and sending concurrent multiple streams of netperf data traffic through one I/O node and arriving at four different data nodes. Data includes four individual links, data bandwidth, and the accumulated data bandwidth. It can reach about 2950MB/sec. Figure-10 shows the result of using seven compute nodes. We can push the bandwidth to 4100MB/sec. Figure-9 and Figure-10 prove that we can gain more bandwidth when more compute nodes are involved in sending networking traffics. This also demonstrates the scaling capability of using the LANL's PaScalBB server I/O infrastructure.

In Figure-11, we verify the advantage of using Linux Ethernet bonding capability. We try two Ethernet bonding algorithm implemented in Linux Kernel: mode-0 and mode-5. Linux Ethernet bonding algorithm "mode-0", named balance-rr or Round-robin policy. It transmits data packets in sequential order from the first available slave through the last. This mode provides load balancing and fault tolerance. Linux Ethernet bonding algorithm mode-5, named balance-tlb or Adaptive transmits load balancing. It supports channel/port bonding that does not require any special switch support. The outgoing data traffic is well distributed according to the current load on each slave link. In-coming data traffic is received by the current slave link. If the receiving slave fails, another slave takes over the MAC address of the failed receiving slave. The purpose of this testing is to figure out a better traffic load balancing algorithm that can accommodate the advantage of parallel file systems used in HPC machines. Our results show that mode-5 (Adaptive transmit load balancing) can obtain 10%-15% more bandwidth compared with mode-0 (a simple Round-robin policy).

From the above results, we can conclude that there is definitely an advantage of using multiple 10-gigabits Ethernet bonding in an I/O node when transferring data through an IB/QDR link. We also learn how to tune 10-Gigabit Ethernet bonding algorithms to come out with the best fit for HPC parallel file system such as the Paranas Panfs ActiverScale Parallel File storage system .

5 CONCLUSIONS AND FUTURE WORKS

We evaluate the bandwidth performance of using IB/SDR/, IN/QDR, and IB/QDR. We also evaluate of various bonding algorithms of using multiple 10-Gigabit Ethernet links. We verify the capability of an I/O node equipped with one IB/QDR and multiple 10-Gigabit Ethernet links. We study the Linux Ethernet bonding algorithms. We observe the scaling capability of an I/O when it handling more network traffics. We figure out a better way of network setup and configuration for LANL's PaScalBB network. We have applied our testing results to LANL's production machines.

As part of the future works, we intend to conduct evaluations on larger test beds, possibly using some available production HPC machines, and studying the impact of new PaScalBB network setups and configuration. We also intend to carry more in-depth studies of applying different network

benchmarking testing, MPI-IO testing, and parallel file system testing.

REFERENCES

[1] Hari Subramoni, Matthew Koop and Dhableswar K. Panda, "Designing Next generation Clusters: Evaluation of Infiniband DDR/QDR on Intel Computing Platforms, HOTI'09 - 17th IEEE Annual Symposium on High-Performance Interconnects

[2] Matthew J. Koop, Wei Huang, Karthik Gopalakrishanan, Dhableswar K. Panda, "Performance Analysis and Evaluation of PCIe 2.0 Quad-Data Rate Infiniband", HOTI'09 -16th IEEE Annual Symposium on High-Performance Interconnects

[3] Infiniband Road map, Infiniband Trade Association, <http://www.infinibandta.org/>

[4] HPC Advisory Council – Network of Expertise, "Interconnect Analysis: 10GigE and infiniband in High Performance Computing, 2009

[5] Munira Hussain, Gilad Shalner, Tong Liu, Onur Celebioglu, "Comparing DDR and QDR Infiniband 11th-generation Dell Powerededge Clusters", DELL Power Solution, 2010 Issue 1

[6] Gary Grider, Hsing-bung Chen, James Nunez, Steve Poole, Rosie Wacha, Parks Fields, Robert Martinez, Paul Martinez, Satsangat Khalsa, "PaScal – A New Parallel and Scalable Server IO Networking Infrastructure for Supporting Global Storage/File Systems in Large-size Linux Clusters", Proceedings of the 25th IEEE International Performance, Computing, and Communications Conference, 2006 (IPCCC 2006). April 2006.

[7] Hsing-bung Chen, Gary Grider, Parks Fields, "A Cost-Effective, High Bandwidth Server I/O network Architecture for Cluster Systems", 2007 IEEE IPDPS Conference

[8] Hsing-bung Chen, parks Fields, Alfred Torrez, "An Intelligent Parallel and Scalable Server I/O Networking

Environment for High Performance Cluster Computing Systems", PAPTA 2008 Conference

[9] OFED – OpenFabrics, <http://www.openfabrics.org>

[10] Mellanox – <http://www.mellanox.com/>

[11] Arista network - <http://www.aristanetworks.com/>

[12] Netperf - <http://www.netperf.org/netperf/>

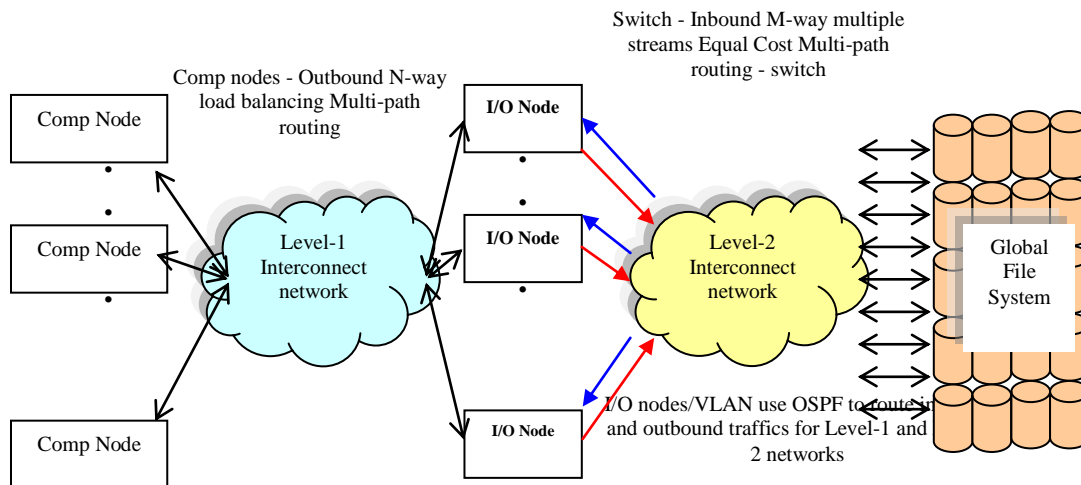


Figure 1: System diagram LANL's PaScalBB Server I/O architecture

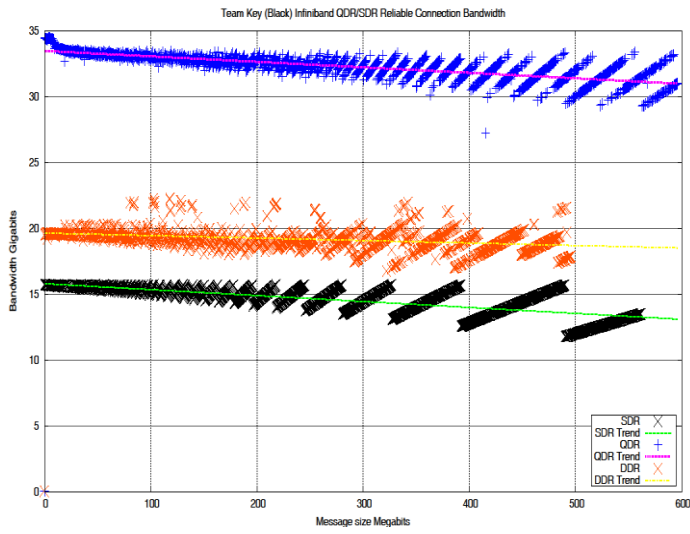


Figure-2: IB/SDR, IB/DDR, and IB/QDR performance testing

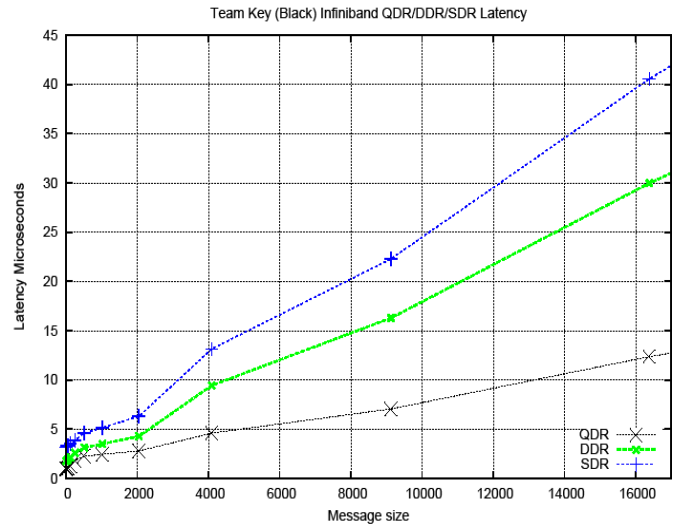


Figure-3: IB/SDR, IB/DDR, and IB/QDR latency testing

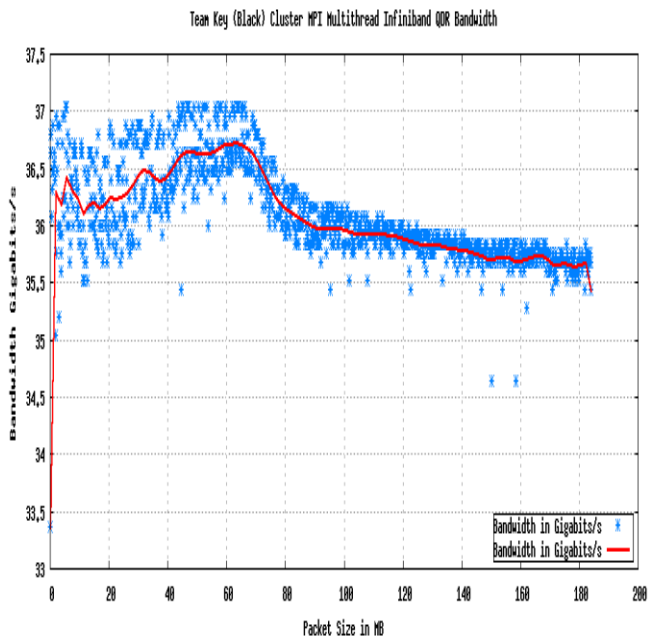


Figure-4: Multithread MPI testing using IB/QDR

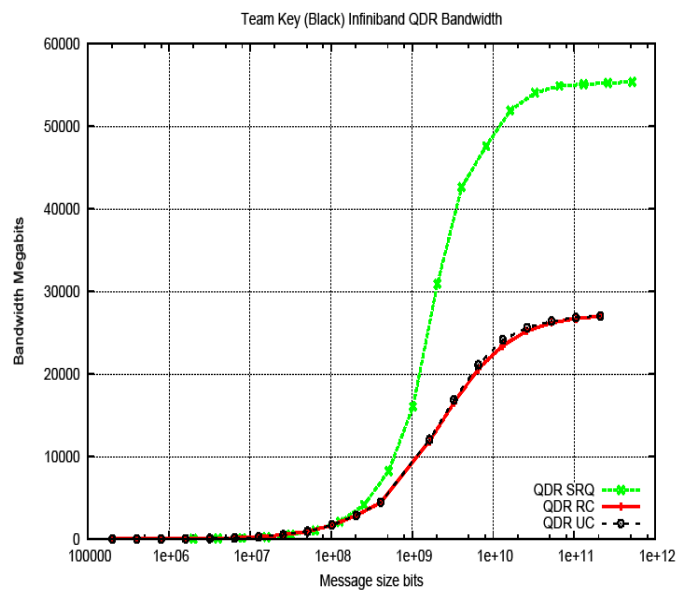


Figure-5: IB/QDR bi-directional bandwidth testing

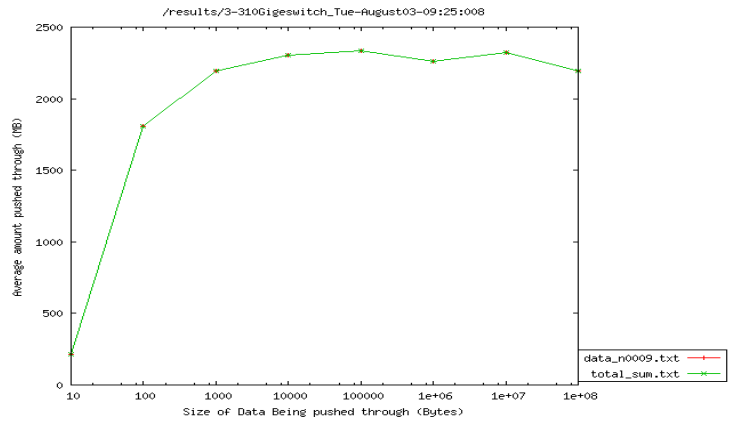
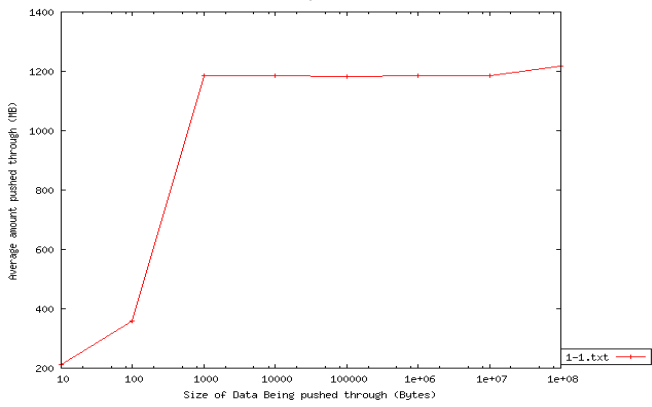


Figure-6: back-to-back one single 10-Gigabit Ethernet testing

Figure-7: Three 10Gigabit Ethernet bonding performance testing

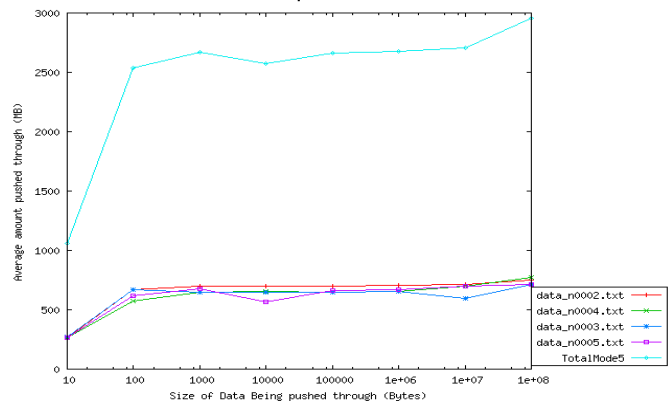
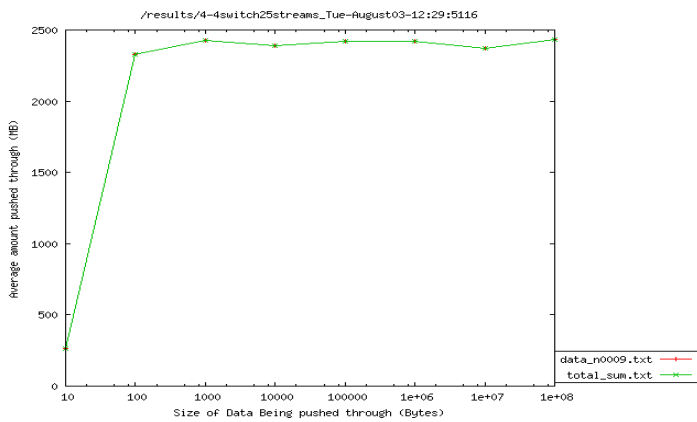


Figure 8: Four 10Gigabit Ethernet bonding performance testing

Figure 9: Using four compute nodes – scaling testing

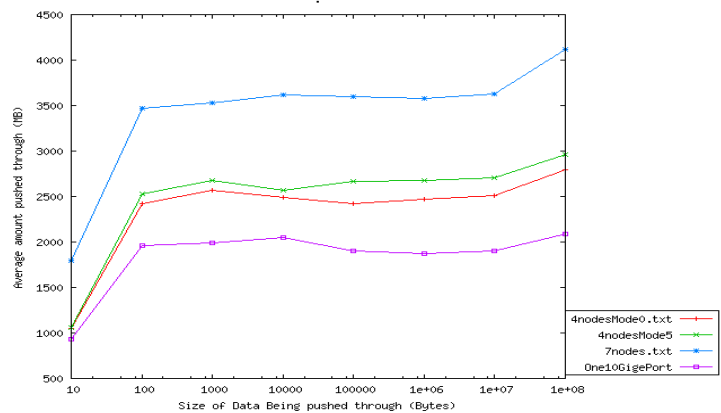
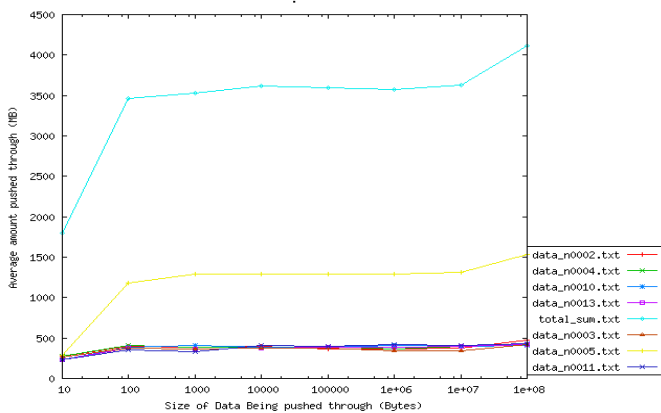


Figure 10: Using seven compute nodes – scaling testing

Figure 11: Linux bonding – mode-0 vs. mode-5 testing

A Set of Microbenchmarks for Measuring OpenMP Task Overheads

James LaGrone¹, Ayodunni Aribuki¹, and Barbara Chapman¹

¹Department of Computer Science, University of Houston, Houston, Texas, USA

Abstract—

Asynchronous tasks make it easy to express the parallelism in a broad variety of computations and are especially useful for writing parallel applications with irregular and/or dynamic workloads. Their introduction into the OpenMP specification has greatly extended the scope of this API. Yet the body of benchmarks using OpenMP tasks remains minimal. The EPCC OpenMP Microbenchmarks provide measurements of overheads incurred by OpenMP constructs in version 2.0-compliant implementations. These microbenchmarks are widely used to explore the behavior of OpenMP constructs on a given platform. To thoroughly test an OpenMP 3.0 implementation, we have extended these microbenchmarks by twenty-one new microbenchmarks that measure overheads incurred by various common uses of OpenMP tasks, including task synchronization. We include evaluations of both commercial and open source implementations of OpenMP tasks on various multicore platforms.

Keywords: OpenMP; asynchronous tasks; benchmarks; runtime

1. Introduction

Asynchronous tasks make it easy to express the parallelism in a broad variety of computations and are especially useful for writing parallel applications with irregular and/or dynamic workloads. Their introduction into the OpenMP [1] specification has greatly extended the scope of this API from a loop-centric model to one that can express the irregular parallelism present in recursive and pointer chasing algorithms. OpenMP is widely used for programming on shared memory systems and in conjunction with MPI on distributed systems.

While there are many benchmarks and applications using OpenMP, MPI, or mixed-mode MPI/OpenMP, only the EPCC OpenMP Microbenchmarks [2], [3] provide measurements of overheads incurred by OpenMP 2.5 constructs. These microbenchmarks are widely used to explore the behavior of OpenMP constructs for an implementation on a given system. We have extended these microbenchmarks with twenty-one new microbenchmarks that measure overheads incurred by various common uses of OpenMP tasks, including task synchronization. We evaluate two implementations of OpenMP tasks, Intel C compiler 11.1 and GNU C compiler 4.6, on various multicore platforms. This will enable a more thorough evaluation of OpenMP 3.0 task implementations.

The remainder of this paper is organized as follows. Following some discussion on related work in Section 2, Section 3 presents an overview of tasks as defined in the OpenMP 3.0 Application Program Interface (API) and some basic concerns regarding its implementation. Sections 4–5 present the details of the microbenchmarks and their results. We conclude and discuss future work in Section 6.

2. Related Work

We have surveyed several benchmarks that use OpenMP for parallel programming on shared memory systems (SMPs), including the hybrid MPI-OpenMP model on SMP clusters. While various OpenMP constructs are included in these benchmarks, the majority rely primarily on the use of parallel loop constructs. Though OpenMP originally targeted loop-centric parallelism, it has since evolved to include task parallelism in version 3.0.

While there are many benchmarks using OpenMP [4]–[10], we are only aware of one set of benchmarks that includes OpenMP tasks [11] written specifically to evaluate OpenMP task implementations via a set of applications that feature regular, irregular, and recursive task parallelism. However, these are not intended to measure implementation overheads.

We present two microbenchmarks for OpenMP derived from the EPCC microbenchmarks [2], [3] and extensions [12] that measure the overhead incurred by implementations of OpenMP prior to version 3.0. One analysis of an implementation of OpenMP tasks [13] used synthetic microbenchmarks, without synchronization, to examine the possibilities of the OpenMP tasking model under various conditions such as task granularity, time between task creation, and common clauses. At the time of this writing, we are unaware of any set of microbenchmarks available for measuring the overhead incurred by use of OpenMP tasking constructs.

3. OpenMP and Tasks

OpenMP [1] is a high-level, explicit programming model for shared memory platforms. It is supported by most commercial and open source compilers and enjoys widespread use in many scientific programming domains. Its latest specification (3.0) moved OpenMP from a purely thread-centric execution model to a more task-centric model. While continuing to use the fork-join parallel execution model, the master thread is now not only responsible for creating a

thread team, but also for generating the set of tasks to be executed by the team. An OpenMP program begins with a single thread of execution which executes the initial (implicit) task in a sequential manner. When any thread encounters an `omp parallel` directive, it creates a team of threads and a set of *implicit* tasks is created and assigned one per thread in the team.

3.1 Tasking Model

Two task-specific constructs, `omp task` and `omp taskwait`, allow the creation and synchronization of tasks. A thread encountering an `task` directive generates an *explicit*, asynchronous task. The `taskwait` or `omp barrier` directives can be used to synchronize tasks. The `taskwait` requires the current task to wait for the completion of all tasks it has generated up to that point in the code. The `barrier` construct defined in previous specifications has been extended to accommodate tasks. Threads encountering a `barrier` must wait until all threads in the current thread team enter the barrier and all tasks created in the parallel region prior to the barrier must complete before any threads may continue. Proper synchronization of tasks is necessary to guarantee the completion of all tasks.

An OpenMP *task* is defined to be a specific instance of executable code and its data environment. The execution of an explicit task may be immediate or deferred, and the task does not have to be executed by the same thread that created it. The execution of a task may be suspended and resumed later. Unless the task is marked `untied`, it must be executed by only one thread and is not subject to possible workstealing. An optional `if` clause, when evaluating to false, causes the encountering task to be suspended and the execution of the new task must begin immediately. Data-sharing attributes are defined using `private`, `firstprivate`, and `shared` clauses similar to `parallel` and `worksharing` constructs. Unless otherwise specified by such a clause or implied by context, the default data-sharing attribute for a `task` is `firstprivate`, meaning the value of the specified data is initialized at the point the `task` directive is encountered and kept private to the task.

3.2 Task Implementation Concerns

Applications with irregular parallelism, like those that are highly recursive or use pointer-chasing algorithms, benefit greatly from asynchronous tasking features in a language. The dynamic nature of tasking makes its implementation much more difficult than the static constructs and avoidable overheads may be introduced by poor implementations, which may limit its scalability. Tasking in OpenMP is a relatively new feature and extensions to the tasking model are likely in the near future. It is possible for the compiler to provide information that could be used to base a runtime solely on tasks [14]. It is therefore critical for the basis of a tasking runtime to be efficient before being extended.

The role of the runtime in a tasking implementation includes creating and scheduling tasks as well as enforcing dependencies between them. Therefore, the efficiency of the runtime implementation has a heavy impact on the performance of task-based applications. Ideally, tasks will be scheduled for execution in a manner that maximizes concurrency while accounting for locality, load imbalance, synchronization, and memory footprint to facilitate better performance. To address these concerns, an implementation should carefully consider how tasks are created, stored, and scheduled for execution, and synchronized.

3.2.1 Queue Organization

Task schedulers are often built using a set of queues for holding tasks that are ready for execution. Possible organizations of queues include a single, centralized queue shared among all threads, queues distributed among the threads, and a hierarchy of queues structured in a tree [15]–[17]. The choice of organization may reflect the desire to target some concerns over others, including ease of implementation, load balancing, contention for queue access, locality of work and data, work stealing, and architectural layout. There are obvious tradeoffs that must be made and integration of the tasking model into an existing OpenMP runtime may preclude some of the more desirable choices.

3.2.2 Task Scheduling

The OpenMP specification does not specify how tasks should be scheduled, but leaves it to the implementation to decide when to execute a task and by which thread. The design of a scheduler should carefully consider the locality of data and load balancing. Tasks using the same data should be scheduled on the same thread, especially on architectures exhibiting non-uniform memory access (NUMA) patterns. The scheduler should also ensure that all threads do close to the same amount of work by dynamically balancing the workload among them. Task scheduling policies can be broadly categorized as work-first or breadth-first [18]. The choice of scheduling policy will likely reflect the choice of queue organization. Because the OpenMP specification allows for workstealing, particular attention may be needed in choosing a thread library to facilitate it.

3.2.3 Task Synchronization

OpenMP specifies the `taskwait` and `barrier` constructs for the synchronization of tasks. Implementations of `omp barrier` must be extended to include unfinished tasks. The `omp taskwait` construct applies only to the children of the parent task, so the implementation must keep track of this parent-child relationship for each task to ensure the proper synchronization and completion of the tasks involved. Implementations employing workstealing may also need to allow for this in the `barrier` and `taskwait` constructs.

4. Methodology

A microbenchmark suite for measuring overheads in OpenMP implementations introduced about a decade ago is often referred to as the EPCC OpenMP Microbenchmarks [19]. This suite of tests provides a framework for evaluating overhead costs incurred by OpenMP implementations by providing a straight-forward method for measuring runtime costs associated with synchronization and loop constructs [20], [21]. The straight-forward measurements it provides have even facilitated implementation improvements [22] by clearly revealing performance problems. Since its introduction it has been updated [2], [3] and other microbenchmarks have followed its lead for measuring overheads in nested parallelism [12] and for evaluating communications in mixed-mode, or hybrid, OpenMP/MPI programming [23]. However, since tasks were introduced in the OpenMP 3.0 specification, these microbenchmarks have not been updated to include them.

The overhead of an OpenMP construct is the difference of the time it takes for a given piece of code to execute sequentially (without using OpenMP) and the composite time for a set of processors to execute the code in parallel with OpenMP [12], [19]. Let T_s represent the sequential execution time of the code block, which is the time it takes to do the actual work of the code block. Letting T_p represent the parallel execution time and n represent the number of processors (or OpenMP threads), nT_p , the composite time of the parallel execution, includes the time spent doing the work of the code block *plus* the time spent constructing and deconstructing the parallel execution environment (i.e., OpenMP overhead, or the execution time NOT spent in the given code block). Algebraically, the total overhead is

$$T_{overhd} = nT_p - T_s \quad (1)$$

and the *per thread* overhead is

$$T_o = T_p - T_s/n. \quad (2)$$

We use a similar framework for the task microbenchmarks that is used in the EPCC Microbenchmarks for OpenMP 2.0 [3]. A reference time, T_s , is calculated by taking the time for the execution of a specific code block used to represent a kernel of actual work. The time T_p for parallel execution is found by timing the execution of the same code block with OpenMP constructs inserted. The overhead, T_{overhd} , incurred by the parallel execution is then calculated according to Equation 1. A significant sample of timings is accrued and the mean, standard deviation, and minimum and maximum values are reported.

OpenMP task constructs can be placed anywhere a legal language statement may appear. However, they are typically placed in `master`, `single`, or `for` constructs in order for each task to be created by one thread. We have constructed tests for each of these methods of task generation as well

as `parallel`. The test using `omp parallel` provides a measure of pure task overhead and the others show the impact of the other typical methods of generating tasks. OpenMP timing routines are used for all tests. The tests in the suite are categorized into tests that involved either `task`, `task firstprivate(data)`, or `taskwait` directives. We have both C and Fortran versions of the microbenchmarks. Code snippets and results herein come from the C versions.

4.1 Measuring Task Overhead

To measure overhead incurred by the `task`, a reference time is found by timing the execution of

```
for (j = 0; j < reps; j++)
    delay( delaylength );
```

enough times to get an adequate sample. This time is divided by *reps* to obtain an average execution time for the construct. The value of *reps* represents the number of times a given construct will be called in the loop of the OpenMP versions of the loop. We have used 30 samples and 10,000 *reps* in our experiments.

The overhead of task generation by the `parallel` construct is measured by timing the execution of the code

```
#pragma omp parallel private(j)
{
    #pragma omp [master|single|for] [untied]
    for (j = 0; j < reps[*ntasks]; j++){
        #pragma omp task
        delay( delaylength );
    }
} /* end parallel */
```

without using a `[master|single|for]` construct for $j \in [0, reps)$. The value of *reps* is chosen to be large enough for the overhead of the enclosing parallel region to be negligible. In this case all threads in the team will execute the `for (j...)` loop. Because all threads will participate in executing the work, the amount of work is equivalent to nT_p . An additional test uses the `untied` clause in the task directive.

Timings of `master`, `single`, and `for` are similar with the respective constructs placed right before the `for (j...)` loop using $j \in [0, reps \times nthreads)$ more iterations to produce an equivalent amount of work per thread. Again, the work in the parallel region should be equivalent to the work done in the serial reference test multiplied by the number of threads. A separate test for each uses the `untied` clause.

4.2 Measuring Overhead of Tasks with First-private Data

Since the use of `firstprivate` data in tasks is not only common but is the default data scope, we felt measuring the overhead of capturing the values of these data could be helpful. These tests are similar those in the `arraybench` set of EPCC microbenchmarks for OpenMP 2.0 [2]. Our tests use small, medium, and large arrays as `firstprivate` data for each of the task generating constructs `master`, `single`, and `for`.

Using `parallel` is not common in this case and is omitted. The timings are calculated in a similar fashion to the `task` tests in Section 4.1.

4.3 Measuring Task Synchronization Overhead

The measuring of the `taskwait` is problematic. In order to get accurate results, the number of `taskwait` constructs encountered needs to be correctly established and the construct only has an affect if there are several generations of tasks created. Therefore, the delay function used for these measurements is recursive and uses arguments for the length of delay and the depth of recursive calls and an additional value to set the branching factor of the tree generated by the recursive calls. The same delay function is used for reference time without any OpenMP directives.

```
void delay(int depth, int delaylength){
    ...
    if ( depth > 1 )
        for ( j = 0; j < BRANCHING; j++){
            #pragma omp task firstprivate\
                (depth, delaylength)
            {
                delay( depth - 1, delaylength );
            }
            #pragma omp taskwait
        }
    /* simulate work */
    for (i=0; i < delaylength; i++) a+=i;
    /* prevent optimization of function */
    if (a < 0) printf("%f \n",a);
    return;
}
```

To achieve the maximum number of `taskwait` directives, each task generated has a `taskwait` associated with it.

For a depth of D and a branching factor of B , the number of tasks, N , generated is

$$N = \sum_{i=1}^D B^{(i-1)} \quad (3)$$

This formula is used to calculate the number of `taskwait` directives encountered by the thread team. The number of tasks is used in place of the `reps` to calculate the timings.

A drawback to this methodology is the resulting one-to-one correspondence of `task` to `taskwait` constructs. This is unavoidable due to the OpenMP specification of `taskwait`. While this leads to `task` overhead in addition to `taskwait` overhead, this would represent a worst case scenario which is unlikely in practice due to the artificial nature of these microbenchmarks.

5. Results

We ran our microbenchmarks on various systems to evaluate several OpenMP implementations. Due to space limitations, we show only results of evaluations on a shared memory multicore system that has dual 2.27 GHz 8-core Intel Xeon Nehalem E5520 processors and 32 GB RAM. Pairs

Construct	Threads				
	1	2	4	8	16
gcc 4.6					
parallel	0.459	1.678	2.348	3.111	3.960
for	0.335	0.552	0.678	1.338	1.187
single	0.334	0.522	0.416	0.792	0.868
icc 11.1					
parallel	0.219	1.443	1.667	2.685	3.797
for	0.015	0.495	0.795	1.156	1.646
single	0.013	0.552	0.805	1.302	1.825

Table 1: Overheads (μsec) as measured in *synbench* of the EPCC Microbenchmarks for OpenMP 2.5 on the 16-core Nehalem.

of cores share 32KB L1 and 256KB L2 caches with each processor sharing 8MB L3 cache. The system was running CentOS 5.5 (final) with a Redhat 2.6.18 series kernel. In our testing we explored the OpenMP implementations in the GNU C compiler 4.6 (*gcc*) and the Intel C compiler 11.1 (*icc*). We present results of our task microbenchmarks for *icc* and *gcc*.

Each timing is an average of three runs, each taking 25 samples, of a given test using different thread team sizes of powers of 2. Minimal activity was observed on the systems before and after each run. The *delaylength* values used were 2000 for the `task` tests and 500 for `omp task firstprivate(a)` and `taskwait`.

5.1 EPCC Results

To obtain a point of reference for the implementation of the constructs used to generate tasks in our tests, we ran the EPCC Microbenchmark [3] tests for `parallel`, `single`, and `for` as found in the synchronization tests known as *synbench* (there is no test for `master`). We ran the tests with a sample size of 20 for each run with a *delaylength* of 500. The averages for 3 runs for the 16-core Nehalem are shown in Table 1. Results from the Opteron were similar and are not shown for space limitations.

We also note that negative overheads may be reported, especially when using one OpenMP thread. Many of these are small enough to be equivalent only a few cycles and can be explained by additional load on the system not measured during the reference time. Some, however, occur consistently over several timings and may reflect optimization employed for OpenMP but not in the sequential portions.

5.2 Task Overhead

We ran tests generating 10,000 tasks to measure `task` overheads. This corresponds to the value of `reps` in the loop creating the tasks and should be adequate to allow the tasking overhead to dominate that of the enclosing `parallel` region. For both *icc* and *gcc*, we see that using the `parallel` and `for` constructs to generate tasks generally incurs less overhead than using `master` and `single` constructs (Figure 1).

The difference increases with the number of threads for both compilers. Note that the scales on the two graphs are different.

We observed no significant differences in results for tests using `master` and `single`, likely because there is little difference in the implementation of the two. The `master` construct requires the master thread to execute the `master` region but has no implicit barrier at its conclusion. The region enclosed by a `single` construct will likely be executed by the first encountering thread but it does have an implicit barrier at the end. We also observed negligible difference between tied and untied tasks. The reason for this could be that the balanced nature of the workload produced in the tests does not lend itself to workstealing. It is also possible that the implementations we considered do not employ workstealing.

In the cases of `master` and `single`, only one thread is generating tasks while the remaining threads in the team initially must wait for something to do. Depending on the task granularity, a single thread may not generate work quickly enough to keep all the threads busy. In contrast, with `parallel` and `for`, the entire thread team is involved in creating tasks. Therefore these results are not entirely surprising. It is interesting to note that in the EPCC results in Table 1, it is the `single` construct in `gcc` that incurs the least overhead. In contrast, the overhead values we obtained for the `task` construct in the `gcc` implementation using the `parallel` and `for` constructs rarely exceeded $2 \mu\text{sec}$ while the `single` construct approached $140 \mu\text{sec}$ of overhead. We speculate that when a single thread is generating tasks, as with `single` and `master` constructs, the rest of the thread team must wait for enough work to be generated to occupy all the threads in executing tasks. The larger the thread team, the longer it will take to generate a workload of this sufficiency. With the entire thread team involved in creating tasks (as with `parallel` or `for`), the threads are more likely to have sufficient tasks to execute.

5.3 Firstprivate Overhead

We measure the overheads of the `task` directive with the `firstprivate` clause on both compilers. Our experiments use dimensional arrays with sizes of 100 bytes for the small test, 2,187 bytes for the medium, and 59,049 for the large test. Because it is so easy to overload the systems, only 500 tasks were generated instead of 10,000 as in Section 5.2. Figure 2 shows the results when a single thread is used to generate tasks that use small, medium, and large arrays as `firstprivate` data. Higher overheads are observed when the numbers are compared with those from experiments without the `firstprivate` clause (Figure 1), but the overheads scale the same way with increasing thread numbers.

We also ran experiments with tasks using the large arrays are generated by the master thread, a single thread (with the `single` directive), and by all the threads using a `for` directive. The results are shown in Figure 3. For both compilers,

we observed that using the master thread to generate the tasks incurs less overhead than the using the `single` and `for` directives. Since we generate 500 tasks with just one of these constructs, the overheads from these constructs can be ignored. We also note that in all other experiments except this one, using the `single` and `master` constructs for generating tasks always showed similar overheads.

5.4 Taskwait Overhead

In the sequential execution of the recursive delay function, the parent-child relationships of the recursive calls can be represented by a tree. This is synonymous with a task tree when `omp task` is present. To examine the effects of `taskwait` with our recursive test, we explored various configurations. Using a branching factor of 100 and a depth of 3 generates a total of 10,101 tasks, very close to the number of tasks generated for tests measuring the overhead of `omp task` in Section 5.2. These values can be altered to provide a deep and narrow tree or a shallow and broad task tree. However, to see the effect the task tree shape may have on the overhead from `taskwait`, we chose one set of tests using parameters that yield approximately the same number of tasks and one set of tests using the same branching factor with varying depths.

In Figure 4, we show results from three tests of the `for` construct using a branching factor of 20 with depths of 3, 4, and 5 yielding 421, 8,421, and 168,421 tasks, respectively. The `gcc` implementation shows nearly the same tree for each, while `icc` does not. For `icc`, the task tree with depth 3 had overhead that measured approximately $10 \mu\text{sec}$, depth 4 measured approximately $15 \mu\text{sec}$, and depth 5 approximately $17 \mu\text{sec}$. This indicates to us that `icc` may be performing optimizations based on the task tree size.

Figure 5 shows results from three tests using different tree structures with similar task counts. The task tree with a depth of 4 and a branching factor of 21 contains 9,724 tasks, the tree with a depth of 6 and a branching factor of 6 has 9,331 tasks and the tree with a depth of 9 and a branching factor of 3 has 9,841 tasks (in accordance with Equation 3). While `gcc` produces similar results regardless of task shape, `icc` again seems to take the task tree into account during execution. The task tree with the most tasks (depth of 9), the deepest and narrowest of the three, measures the lowest overhead. A task scheduling policy may also exploit a task tree generated in recursive algorithms differently than the task structure generated in a loop (as in Section 5.2).

6. Conclusions and Future Work

While it is easy to make comparisons with these data, it should be noted that low overhead of a single construct is not necessarily indicative of the performance of an entire OpenMP implementation. These microbenchmarks are designed to identify possible inefficiencies of individual constructs. In a realistic application, the manner in which

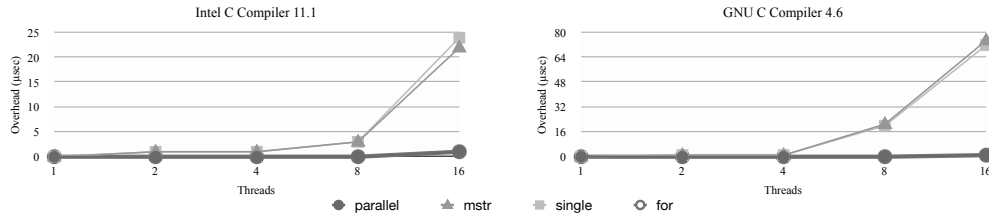


Fig. 1: Comparison of overheads incurred by task generation using `parallel`, `single` and `for` on the 16-core Nehalem machine.

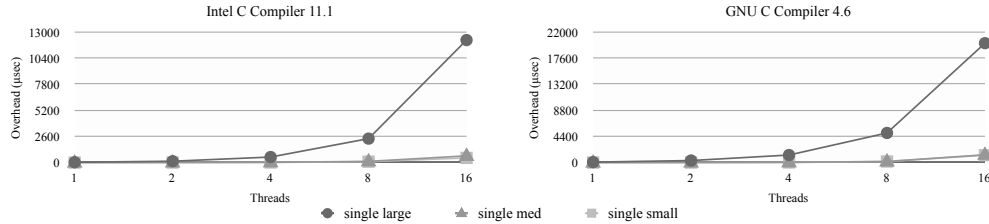


Fig. 2: Comparison of overhead incurred by using `single` with small, medium, and large arrays for data in a `firstprivate` clause using the 16-core Nehalem.

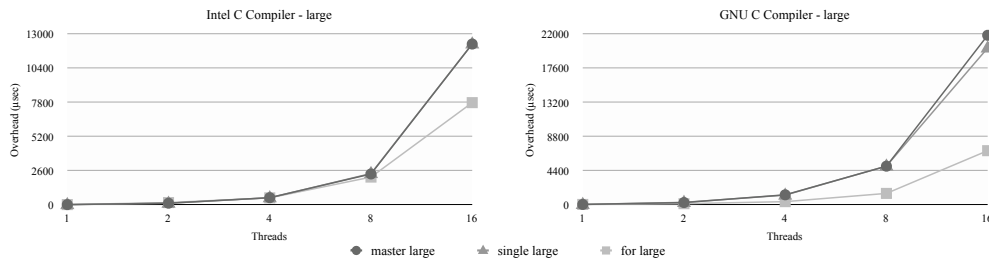


Fig. 3: Comparison of using large arrays in the `firstprivate` tests on the 16-core Nehalem.

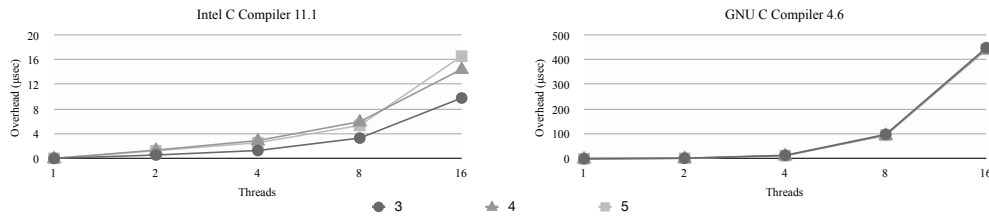


Fig. 4: Overheads for `taskwait` using a branching factor of 20 while varying task graph depths with 3, 4, and 5. These depths yield 421, 8,421, and 168,421 tasks, respectively.

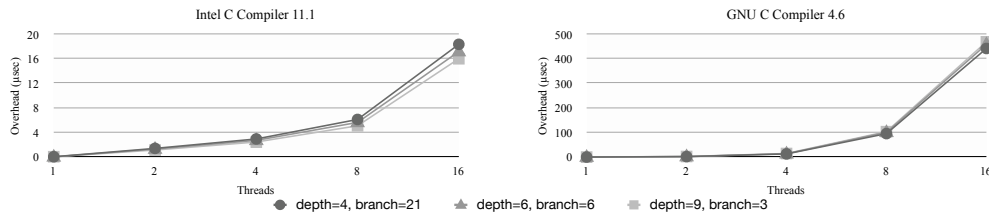


Fig. 5: Overheads for `taskwait` using a branching factor of 20 while varying task tree configurations with similar numbers of tasks.

various constructs work together is much more indicative of the overall efficiency of the implementation as a whole than its performance on these microbenchmarks.

However, it could be inferred that when choosing a task generating construct, `for` should be the first choice. Having the entire thread team (rather than a single or master region) share in the work of task generation appears to yield less latency in execution. The use of parallel seems unlikely in practice and is included merely for evaluation purposes. While we did not test the granularity of the work in tasks, the use of large amounts of `firstprivate` data should be avoided, if possible.

We also found that these tests are very sensitive to system load. Minor fluctuations can be seen at times in the values obtain. It is therefore prudent to run the microbenchmarks multiple times to obtain reliable results. With this in mind, these test should provide a good starting point for evaluating an OpenMP tasking implementations as a means of its improvement.

These microbenchmarks reveal some possible limitations in current implementations. Our next step is to use these microbenchmarks to analyze the OpenMP runtime we have under development to assess any possible improvements that can be made. Both C and Fortran versions will be made available at <http://www.cs.uh.edu/~hpctools>.

In the very near future a new version OpenMP API is likely to be ratified which will include some enhancements to the tasking model. At that time the gap will widen once again between the available and needed benchmarks for evaluation of OpenMP implementations. However, the inclusion of OpenMP tasks in many of the existing benchmarks is difficult or impossible with the current tasking model which makes no accommodations for data dependencies between among tasks. We intend to look into developing new benchmarks or updating existing ones to use OpenMP tasks. We will also look into designing extensions to the OpenMP programming model to facilitate the use of tasks in current codes.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0833201 and Grant No. CCF-0917285, and the Texas Space Grant Consortium. As always, we are indebted to the HPCTools research group at the University of Houston for their help and collaboration.

References

- [1] *OpenMP Application Program Interface Version 3.0*, May 2008.
- [2] J. M. Bull and D. O'Neill, "A microbenchmark suite for OpenMP 2.0," *SIGARCH Comput. Archit. News*, vol. 29, no. 5, pp. 41–48, 2001.
- [3] F. Reid and J. Bull, "OpenMP microbenchmarks version 2.0," *EWOMP 2004*, p. 63, 2004.
- [4] "NAS Parallel Benchmarks," [Online]. Available: <http://www.nas.nasa.gov/Resources/Software/npb.html>.
- [5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.
- [6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, 4-6 2009, pp. 44–54.
- [7] "The Sphinx Parallel Microbenchmark Suite," [Online]. Available: <https://computation.llnl.gov/casc/sphinx/>.
- [8] "ASC Sequoia Benchmark Codes," [Online]. Available: <https://asc.llnl.gov/sequoia/benchmarks/>.
- [9] "HPC Challenge Benchmark," [Online]. Available: <http://icl.cs.utk.edu/hpcc/>.
- [10] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady, "SPECComp: A new benchmark suite for measuring parallel computer performance," *OpenMP Shared Memory Parallel Programming*, pp. 1–10, 2001.
- [11] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé, "Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP," in *38th International Conference on Parallel Processing (ICPP '09)*, IEEE Computer Society, Vienna, Austria: IEEE Computer Society, September 2009, p. 124–131.
- [12] V. Dimakopoulos, P. Hadjidoukas, and G. Philos, "A microbenchmark study of OpenMP overheads under nested parallelism," in *IWOMP '08: OpenMP in a New Era of Parallelism*, ser. Lecture Notes in Computer Science, R. Eigenmann and B. de Supinski, Eds. Springer Berlin / Heidelberg, 2008, vol. 5004, pp. 1–12.
- [13] X. Teruel, C. Barton, A. Duran, X. Martorell, E. Ayguadé, P. Unnikrishnan, G. Zhang, and R. Silvera, "Openmp tasking analysis for programmers," in *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*. ACM, 2009, pp. 32–42.
- [14] T.-H. Weng and B. Chapman, "Implementing OpenMP using dataflow execution model for data locality and efficient parallel execution," in *Proceedings of the 7th workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS-7)*. IEEE Press, 2002.
- [15] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé, "Support for OpenMP tasks in Nanos v4," in *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. ACM, 2007, pp. 256–259.
- [16] M. Korch and T. Rauber, "A comparison of task pools for dynamic load balancing of irregular algorithms," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 1, pp. 1–47, 2004.
- [17] C. Addison, J. LaGrone, L. Huang, and B. Chapman, "OpenMP 3.0 tasking implementation in OpenUH," in *Open64 Workshop in conjunction with the International Symposium on Code Generation and Optimization*, 2009.
- [18] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *Proceedings of the 4th IWOMP*, R. Eigenmann and B. R. de Supinski, Eds., vol. 5004/2010, May 2008, pp. 100–110.
- [19] J. M. Bull, "Measuring synchronisation and scheduling overheads in OpenMP," in *Proceedings of First European Workshop on OpenMP*, 1999, pp. 99–105.
- [20] V. Dimakopoulos, E. Leontiadis, and G. Tzoumas, "A portable C compiler for OpenMP V. 2.0," in *Proc. of the European Workshop on OpenMP (EWOMP'03)*, Aachen, Germany, 2003.
- [21] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "OpenUH: an optimizing, portable OpenMP compiler: Research articles," *Concurr. Comput.: Pract. Exper.*, vol. 19, no. 18, pp. 2317–2332, 2007.
- [22] R. Nanjgowda, O. Hernandez, B. Chapman, and H. Jin, "Scalability evaluation of barrier algorithms for OpenMP," *Evolving OpenMP in an Age of Extreme Parallelism*, pp. 42–52, 2009.
- [23] J. Bull, J. Enright, and N. Ameer, "A Microbenchmark Suite for Mixed-Mode OpenMP/MPI," *Evolving OpenMP in an Age of Extreme Parallelism*, pp. 118–131, 2009.

SESSION

FAULT-TOLERANT SYSTEMS + FAULT DETECTION METHODS AND TOOLS

Chair(s)

TBA

Relentless Computing: Enabling fault-tolerant, numerically intensive computation in distributed environments

Lucas A. Wilson and John A. Lockman III

Texas Advanced Computing Center, The University of Texas at Austin, Austin, Texas, U.S.A.

Abstract—*This paper suggests a novel computational paradigm for solving numerically intensive problems on a distributed infrastructure. We detail the basic functionality of this new paradigm, its ability to recover from host loss without requiring a complete restart of the code, and how it could allow for many heterogeneous participants to solve a single, large-scale computational problem. We provide results from a small demonstration run as well as provide avenues for future research.*

Keywords: Volunteer Computing, Distributed Computing, Fault Tolerance, Distributed Hash Tables

1. Introduction

Computer-based simulation and modeling is becoming critical for driving scientific breakthrough and discovery. As the sensitivity and scale of simulations increase, the computational requirements and time-to-solution also rises. Unfortunately modern hardware – although much improved over technologies of several years ago – does not provide researchers with a stable execution platform for simulations requiring weeks or months of computation to complete, and is extremely expensive to deploy in large-scale, tightly-coupled environments. As a result, computer-based simulation for scientific discovery has remained limited to those researchers who have access to high-performance systems at Universities and National Laboratories.

Fully-distributed volunteer computing models, such as the Berkeley Open Infrastructure for Network Computing (BOINC)[1], have provided a means of performing massive-scale computation on a limited subset of problems involving limited-to-no data sharing among participants. While BOINC and similar volunteer computing models have successfully computed millions of hours of user code, the types of problems that can be executed in such an environment are limited and cannot fully enable new scientific breakthroughs.

Distributed Hash Table (DHT) implementations, such as Chord[2], Pastry[3], Kademlia[4], and others[5][6][7][8], provide mechanisms for storing key/value pairs in a decentralized fashion, preventing the failure of any single participant from killing the entire hash table. DHTs are commonly used in distributed file systems[9][10], peer-to-peer file sharing systems[11], and domain name services[12][13].

Volunteer computing models can harness the untapped computing potential of millions of part-time citizen scien-

tists. We propose a system that would couple this potential with the innately fault-tolerant nature of DHTs, allowing for the execution of programs for extremely long periods of time, with built-in failure recovery in the event any set of participants was unable or unwilling to continue contributing. Additionally, proper data partitioning would allow for problems requiring more onerous data sharing among participants to be executed, increasing the potential for more scientific discoveries.

This work describes a new computational paradigm: Relentless computing. With Relentless Computing, traditionally tightly-coupled, numerically-intensive parallel computations can be performed in a decentralized, distributed environment with high fault-tolerance. So long as any single participant and the initial data are present to the system, computation will continue. We will provide a basic description of Relentless Computing, how code is generated and managed, and how global shared memory is implemented through DHTs. We will also provide results from a test case solving a partial differential equation (PDE) using finite differences, as well as outline avenues for future research.

2. Related Work

While existing distributed computing systems, such as BOINC[1], have been extremely useful for executing completely data-parallel computations, such as Monte Carlo and parametric sweeps, they are ill-suited for handling single, large-scale computations that require data sharing among participants. On the other side of the spectrum, Adaptive-MPI (AMPI)[14], which is based on the Charm++ framework[15], allows for the creation of medium-grained virtualized processes which can be overloaded on a single physical processor in order to overlap computation and communication. AMPI does provide many facilities similar to Relentless Computing, including the ability to shrink/expand the number of computational participants and checkpointing of virtual processors to disk. However, AMPI is not designed to handle code written in different languages, to recover from near-catastrophic node failure without the use of a restart file, or to allow very fine-grained parallelism that enables contributions from low-power participants (e.g. "smart" phones, tablets, netbooks, portable computers) without these devices adversely affecting the overall performance of the system.

GRID-GUM[16] implements Glasgow Parallel Haskell[17] on top of the Globus Toolkit[18], making

use of MPICH-G2[19] for handling the underlying process communications. While this method helps to abstract the parallel computation away from the programmer, the use of MPICH-G2 makes participant shrink/expand nearly impossible, and does not allow for fault-tolerance in the event of near-catastrophic node failure.

The Partitioned Global Address Space (PGAS) model, on top of which languages such as X10[20], Chapel[21] UPC[22] and others[23][24][25] are built, provides a method of abstracting a virtual shared memory platform on top of distributed memory architectures. Currently, PGAS languages running on distributed memory clusters rely on existing message-passing methods, such as MPI[26], to handle the cross-node communication that synchronizes virtual global memory and migrates process threads. Because of this, the fault-tolerance and dynamic capabilities of PGAS languages are limited to the capabilities of the underlying communications framework, of which little currently exists.

Global Arrays[27] provide yet another way of abstracting shared memory on top of a distributed memory architecture. However, memory synchronization is once again dependent on an existing communication framework, either MPI or ARMCI[28], which provide little-to-no capability to shrink/expand the participant pool dynamically at runtime, seamlessly recover from near-catastrophic host failure, or efficiently function over high-latency, low-bandwidth networks.

TStreams (also called Concurrent Collections (CnC)[29]) provide a model of describing computation in terms of serial execution components and data-flow specifications[30], in much the same way that Relentless Computing does. While TStreams provides facilities for creating static checkpoints, we are not aware of any particular implementation of this model that provides fault tolerance that enables continuing execution in the face of hardware failures, or that is designed with high-latency, low-bandwidth interconnects in mind. TStreams is also not specifically designed to incorporate low power consumer devices into the participant pool, or to enable execution on fickle participants that may only allow for the use of a fraction of total cycles to be consumed, for limited amounts of time.

The Linda coordination language allows for the separation of coordinate and computation by placing information into an external tuple-space data store, allowing computation from multiple languages to interact[31]. Fault tolerance mechanisms have also been proposed for Linda[32]. Much of the published work found by the authors on Linda is over a decade old, and many of the principles in Linda are incorporated into Relentless Computing. Relentless computing has been designed from the start to work on highly distributed, Internet-connected devices of varying computation capability with the ability to shrink/expand the participant pool at will, as well as recover from near-catastrophic failures.

3. The Relentless Computing Model

The Relentless Computing model seeks to leverage the untapped computing potential of various hardware resources, all connected to the Internet by some mechanism (hardware, wireless, cellular). The use of high-latency, low-bandwidth network connections requires that computation remain limited to highly partitioned, small pieces of data to allow for reasonable read/write from/to the DHT. Programs are written as codelets (self-contained pieces of code) chained together by data dependency. When a hardware resource volunteers to participate in the solving of a particular problem, the solution is sought in a bottom-up fashion, with the participant seeking to complete the result first. If a participant is unable to build the result, it steps up the dependency chain until a data part that can successfully be computed is found. Code written for this environment are built in two pieces: (1) A set of multi-language codelets, which can interact with each other through global shared memory implemented in the DHT, and (2) a descriptive framework that determines the order in which these codelets are to run, and the data dependencies that chain them together.

3.1 Codelets

Each codelet is a self-contained piece of code that performs a set of sequential load/compute/store operations. Because there is no direct interaction between codelets (i.e. each codelet is independent of each other, with interaction performed through data sharing), codelets can be constructed in different languages. This allows for development teams to be able to work in the languages that members are most comfortable with, without worrying about the issues involved in coupling different languages together in traditional software development. Additionally, reducing the codelet size and its associated data dependencies allows for out of order execution of codelets to occur, provided the input data is available.

Codelets can be precompiled in a compilable language, so long as they are capable of executing on the participant hardware. Scripting languages can also be utilized, assuming a mechanism exists to execute the script code from the compute daemon. Runtime environments for some scripting languages (e.g. JavaScript via SpiderMonkey[33], Python[34], Lua[35], LISP via ECL[36], etc.) could be embedded directly into the compute daemon to allow the daemon to compute results directly.

3.2 Memory Management

Relentless computing environments (RCEs) create a global shared memory space from which codelets can read data and to which codelets can write data. This global shared memory space is implemented as key/value tuples to the DHT, with the key replacing the variable name/address, and the value representing the stored data. This is similar to the tuple-space data storage methods employed in

the coordination language Linda[31]. In order to eliminate possible side-effects associated with uncoordinated writes to memory each codelet must be deterministic, with one set of inputs guaranteed to produce the same output. This allows for the possibility of multiple resources executing the same codelet instance (perhaps because both attempted to solve it simultaneously, or knowledge that a particular data point had already been computed was temporarily lost). In order to reduce memory bloat, entries in the DHT will be given a lifetime (e.g. 24 or 48 hours). During the lifetime of the data, codelets can use that information as input for other computation. After that time the data will be deleted, and any participant requiring that particular key/value pair will be required to recompute it.

3.3 Problem Description Framework

In order to chain together multiple codelets and have them interact with the DHT-implemented global shared memory space, a problem description framework must exist that allows Relentless Computing daemons to determine which codelets to run and with which data to run them. This framework must provide the ability to easily define inputs and outputs, as well as specify the particular codelet to execute for each input/output set. Additionally, a result component must be specified so that participants know which data elements are considered final, providing a starting point from which execution can begin. One choice for this is to create an extensible framework language with the Extensible Markup Language (XML)[37]. Not only does XML provide the extensibility to add new features and constructs easily, it is well accepted and understood by the community and is easily compressible, allowing for faster transmission between compute daemons.

A potential problem description for solving the 1-dimensional heat equation using the Forward in Time, Central in Space (FTCS) method may look like Figure 1. In this case, we have assumed that the compute daemon can natively interpret JavaScript code and that the boundary values have already been inserted into the system. Each time the codelet is executed, it requires three inputs (denoted by the `depends-on` tags): the (l)eft, (m)iddle, and (r)ight values from the previous timestep and outputs a single value $u[x][t]$. Function parameters are linked to values in the `depends-on` tags in order from top to bottom, so parameter `l` is associated with $u[x-1][t-1]$, `m` is associated with $u[x][t-1]$, and `r` is associated with $u[x+1][t-1]$.

When a user submits a job containing a problem description and codelets to an RCE, that problem description will then spread across the network to various participants using a gossip protocol [38]. Once other participants are made aware of the new problem, they can begin solving it as well. Each participant – responsible for both starting codelets as well as participating in the DHT – has no advance knowledge about the current state of the problem. In order to

```
<problem name=heat_transfer>
  <codelet name='finite_diff'>
    <result/>
    <source lang='javascript'>
      <![CDATA[
        function finDiff(l, m, r) {
          return m + 0.25*(1 - 2*m + r);
        }
      ]]>
    <parameter name='x' range='0..99' />
    <parameter name='t' range='1..99' />
    <depends-on name='u[x-1][t-1]' />
    <depends-on name='u[x][t-1]' />
    <depends-on name='u[x+1][t-1]' />
    <output name='u[x][t]' />
  </codelet>
</problem>
```

Fig. 1: Potential Problem Description

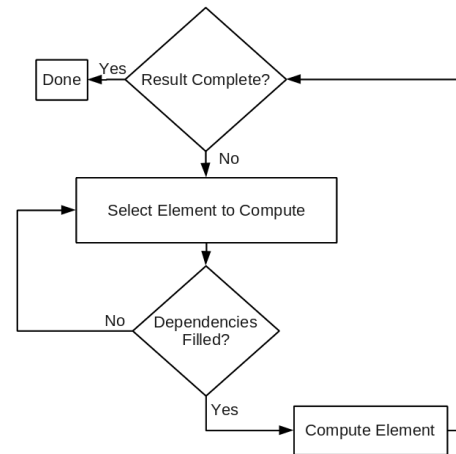


Fig. 2: Process Diagram for RCE daemon

determine which codelet to run while avoiding duplication of effort, the computing daemon parses the work-flow in a bottom-up fashion, beginning with the result codelet and working its way back up the dependency chain until a codelet that is capable of being executed (but that has yet to be executed) is discovered. Once a codelet has been executed and the resulting data stored in the DHT, the compute daemon begins again with the result codelet, working back up the dependency chain in order to take advantage of other more recent dependencies that may have been computed in parallel. The process diagram of an RCE daemon is shown in Figure 2.

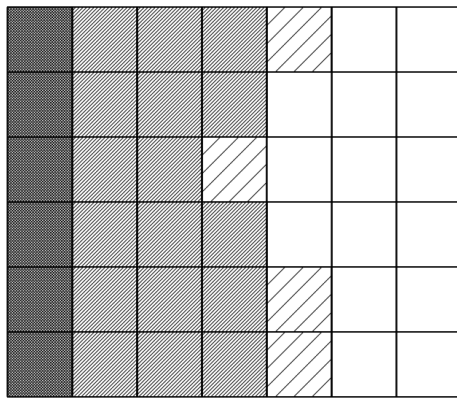
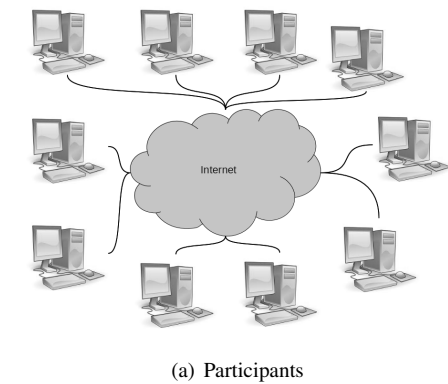


Fig. 3: Active Compute Collective

3.4 Managing Node Failure

In any large-scale distributed environment, node failure is a constant risk that cannot be ignored. In most high-performance machines and with typical multi-thread/process computing paradigms (Pthreads, OpenMP[39], MPI[26], Parallel Virtual Machine (PVM)[40], PGAS languages[20][21][22][23][24][25]), the loss of a host or process results in the simultaneous aborting of all processes associated with a problem. This error handling method may be sufficient in environments with relatively high uptime guarantees. However, distributed volunteer computing environments provide no such guarantee. As a result, more effective fault-tolerance mechanisms must be employed.

DHTs are by their nature relatively fault-tolerant. So long as any single host remains, part of the DHT still exists. The loss of any single participant does not destroy the entire table. The compute daemons of the proposed system would also be the DHT participants, each locally storing part of the hash table in addition to volunteering computational cycles.

Figure 3 shows an example of a volunteer collective working on a particular problem, for example a finite difference problem with a 3-point central difference in the

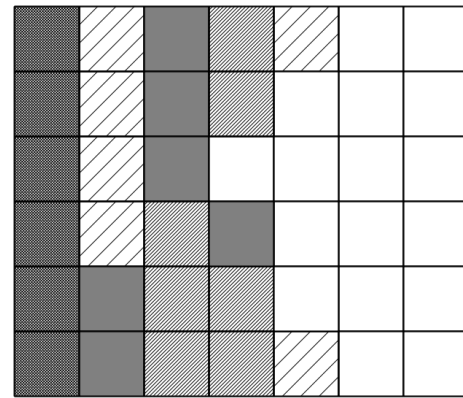
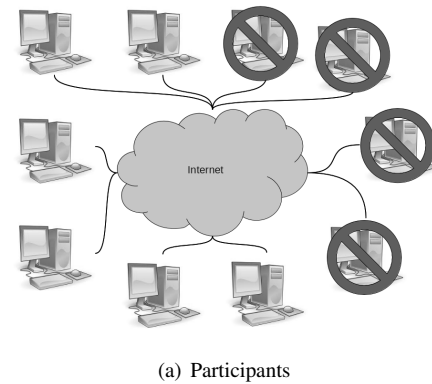


Fig. 4: Compute Collective After Node Failures

space dimension (vertical axis) and a forward difference in the time dimension (horizontal axis). In this example, many participants (Figure 3(a)) are working on various parts of the problem (represented by the logical matrix in Figure 3(b)). The leftmost column values are the initial data, the densely striped values are already computed pieces, and the sparsely striped values are those pieces that can currently be computed with the data that already exists. The boundary values are not shown in this illustration, but can be considered initial data if the boundaries are constant.

If, before the next step can be computed, several of the participants fall offline (Figure 4(a)), some of the data stored in the DHT would be lost (Figure 4(b), gray values). In this case, participants would be unable to compute all of the values that were previously possible due to loss of input data. Instead, they would continue on as though those values never existed, in some cases needing to return to the input data in order to compute the necessary intermediate values.

4. Experiment Setup

In order to test the viability of the proposed paradigm in solving a traditional numerically intensive problem, a prototype RCE daemon was written in Python using the Entangled [41] library to perform the base DHT operations. The

Table 1: Breakdown of participant contributions

Participant	Contributions	Percentage
0	2841	27.3
1	2629	25.3
2	2462	23.7
3	2468	23.7
Total	10400	100

prototype daemon was written to solve the one-dimensional heat equation using the FTCS method:

$$u_x^{t+1} = u_x^t + r(u_{x-1}^t - 2u_x^t + u_{x+1}^t), \quad r = \frac{\alpha \Delta t}{\Delta x^2}$$

In this case, each element at time $t + 1$ is computed by looking at the corresponding values of itself and its neighbors at time t , meaning each element computed needs three inputs from the previous time-step.

For this test, a constant heat source was placed in the leading boundary ($x = -1$), while the trailing boundary was set to 0. The initial ($t = 0$) temperature of the system is set to 0. Experiments were run for a 100x100 case (100 spatial units for 100 time-steps). For this experiment, 4 participants (2 nodes with 2 participants each) were used. In order to test the ability of the system to handle new participants joining mid-computation, participant 0 was initially alone, with participant 1 added next, followed by participants 2 and 3. A communication error (host disconnected from then reconnected to the Internet) was introduced several minutes into the simulation to test the RCE's failure recovery capability.

5. Results

Data collected from a 4 participant run were graphed based on times at which data was stored in the DHT, with both solution values and contributing participant recorded. In Figure 5, the left panels show the solution values to the heat problem, while the right panels show which participant contributed that particular element to the solution. As can be seen, the overall problem was solved in non-linear order, with some sections of the solution growing faster than others. Additionally, the right panels show that work was well distributed, with participant 0 naturally performing more operations than the others, and participants 2 and 3 nearly equal (they joined the computation at the same time) (see Table 1). Table 1 also shows total contributions of 10,400, while the total number of cells in the 100x100 system is 10,000. This means that 400 elements, or 4%, were recomputed for various reasons including simultaneous attempts to calculate a specific element, and the test communication failure that prevented participants from querying the full DHT.

Of the 400 elements that were recomputed, 356 were recomputed only once, while 22 were recomputed twice. Times between recomputations varied widely, with a maximum time between first element computation and final

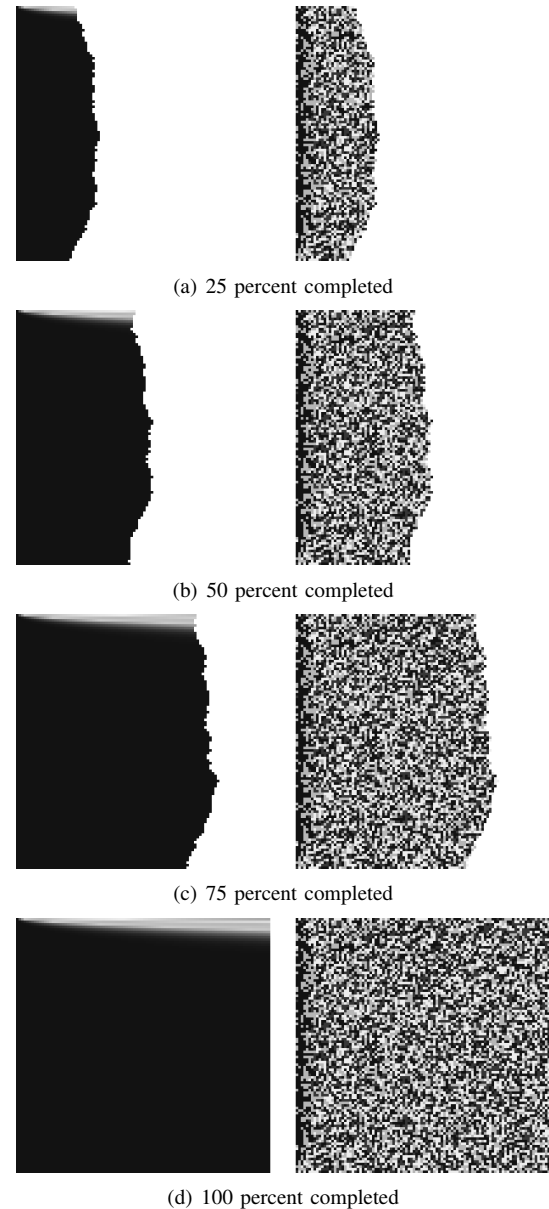


Fig. 5: RCE computing solution to 1-D heat equation

recomputation of 683.62 seconds (see Table 2 and Figure 6). The average recomputation time was 104 seconds, with a standard deviation of 119 seconds.

6. Conclusions

We have proposed a novel computational paradigm called Relentless Computing. This new paradigm allows users to develop codes that solve numerically intensive problems on more disparate and distributed resources, as well as provide the ability for dynamic expanding and shrinking of the participant pool. This model of computation provides fault tolerance, in that the code will continue to execute

Table 2: Time Between Recomputations

Metric	Time (secs.)
Max	683.62
Average	104.78
Std. Dev.	119.12
Median	56.82

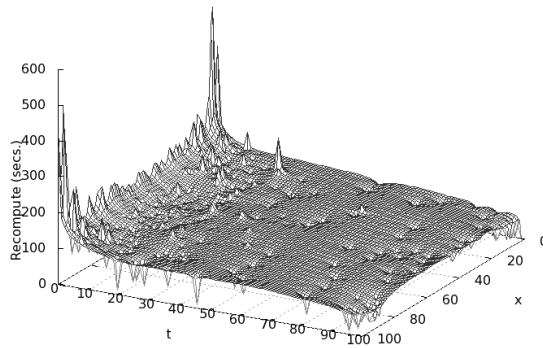


Fig. 6: Times Between Initial and Final Recomputation for Each Element

so long as the initial data and a single participant remain online. Early experimental results have shown that this paradigm provides a relatively simple way for computing numerically intensive problems in a distributed, decentralized, dynamic, and fault-tolerant fashion without requiring excessive work from the programmer. Future work could include development of a standardized problem description framework language, porting of traditional scientific codes to this paradigm, further optimization of the RCE daemon to be more computationally efficient, and inclusion of multiple language runtime environments into the daemon to allow for codelets written in script and interpreted languages to be executed directly by said daemon.

References

- [1] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, ser. GRID '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 4–10. [Online]. Available: <http://dx.doi.org/10.1109/GRID.2004.14>
- [2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160. [Online]. Available: <http://doi.acm.org/10.1145/383059.383071>
- [3] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *MIDDLEWARE 2001*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001, vol. 2218/2001, pp. 329–350.
- [4] P. Maymounkov and D. Mazières, "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS '01. London, UK: Springer-Verlag, 2002, pp. 53–65. [Online]. Available: <http://portal.acm.org/citation.cfm?id=646334.687801>
- [5] K. Aberer, P. Cudré-Mauroux, A. Datta, Z. Despotovic, M. Hauswirth, M. Puceva, and R. Schmidt, "P-Grid: a self-organizing structured P2P system," *SIGMOD Rec.*, vol. 32, pp. 29–33, September 2003.
- [6] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz, "Tapestry: a resilient global-scale overlay for service deployment," *Selected Areas in Communications, IEEE Journal on*, vol. 22, no. 1, pp. 41 – 53, January 2004.
- [7] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," *SIGCOMM Comput. Commun. Rev.*, vol. 31, pp. 161–172, August 2001.
- [8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, October 2007.
- [9] P. Druschel and A. Rowstron, "PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility," *Hot Topics in Operating Systems, Workshop on*, vol. 0, p. 0075, 2001.
- [10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 202–215.
- [11] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips, "The BitTorrent P2P File-Sharing System: Measurements and Analysis," in *Peer-to-Peer Systems IV*, ser. Lecture Notes in Computer Science, M. Castro and R. van Renesse, Eds. Springer Berlin / Heidelberg, 2005, vol. 3640, pp. 205–216.
- [12] V. Ramasubramanian and E. G. Sirer, "The design and implementation of a next generation name service for the internet," in *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, ser. SIGCOMM '04. New York, NY, USA: ACM, 2004, pp. 331–342. [Online]. Available: <http://doi.acm.org/10.1145/1015467.1015504>
- [13] Y. Doi, "DNS Meets DHT: Treating Massive ID Resolution Using DNS Over DHT," *Applications and the Internet, IEEE/IPSJ International Symposium on*, vol. 0, pp. 9–15, 2005.
- [14] C. Huang, O. Lawlor, and L. V. KalÁl, "Adaptive MPI," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, L. Rauchwerger, Ed. Springer Berlin / Heidelberg, 2004, vol. 2958, pp. 306–322.
- [15] L. V. Kale and S. Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 91–108.
- [16] G. M. A. D. Al Zain, P.W. Trinder and H.-W. Loidl, "Managing heterogeneity in a grid parallel Haskell," *Scalable Computing: Practice and Experience*, vol. 7, pp. 9–25, September 2006.
- [17] J. G. Hall, C. Baker-Finch, P. Trinder, and D. J. King, "Towards an operational semantics for a parallel non-strict functional language," in *Proceedings of the International Workshop on the Implementation of Functional Languages (IFL'98)*, September 1998. [Online]. Available: <http://mcs.open.ac.uk/djk26/apset/transitionsystem.ps>
- [18] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of High Performance Computing Applications*, vol. 11, pp. 115–128, 1997.
- [19] N. T. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-enabled implementation of the Message Passing Interface," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 551 – 563, 2003, special Issue on Computational Grids. [Online]. Available: <http://www.sciencedirect.com/science/article/B6WKJ-48BKT9V-1/2/6462834e0f7d0175d57043bbf3df8a80>
- [20] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented program-*

- ming, systems, languages, and applications, ser. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 519–538.
- [21] B. Chamberlain, D. Callahan, and H. Zima, “Parallel Programmability and the Chapel Language,” *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1286120.1286123>
- [22] T. El-Ghazawi and L. Smith, “UPC: unified parallel C,” in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06. New York, NY, USA: ACM, 2006.
- [23] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken, “Titanium: a high-performance java dialect,” *Concurrency: Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.
- [24] E. Allen, D. Chase, J. Hallett, V. Luchango, J.-W. Maessen, S. Ryu, G. S. Jr., and S. Tobin-Hochstadt, “The Fortress Language Specification,” 2008. [Online]. Available: <http://labs.oracle.com/projects/plrg/Publications/fortress.1.0.pdf>
- [25] R. W. Numrich and J. Reid, “Co-array Fortran for parallel programming,” *SIGPLAN Fortran Forum*, vol. 17, pp. 1–31, August 1998.
- [26] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MA, USA: MIT Press, 1994.
- [27] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, “Global arrays: a portable “shared-memory” programming model for distributed memory computers,” in *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, ser. Supercomputing '94. New York, NY, USA: ACM, 1994, pp. 340–349.
- [28] J. Nieplocha and B. Carpenter, “ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems,” in *Proceedings of the 11 IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. London, UK: Springer-Verlag, 1999, pp. 533–546. [Online]. Available: <http://portal.acm.org/citation.cfm?id=645611.662053>
- [29] K. Knobe, “Ease of use with concurrent collections (CnC),” in *Proceedings of the First USENIX conference on Hot topics in parallelism*, ser. HotPar'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 17–17. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1855591.1855608>
- [30] K. Knobe and C. D. Offner, “TStreams: How to Write a Parallel Program, Tech. Rep. HPL-2004-193,2004. [Online]. Available: <http://www.hpl.hp.com/techreports/2004/HPL-2004-193.pdf>
- [31] N. Carriero and D. Gelernter, “Linda in context,” *Commun. ACM*, vol. 32, pp. 444–458, April 1989. [Online]. Available: <http://doi.acm.org/10.1145/63334.63337>
- [32] D. E. Bakken and R. D. Schlichting, “Supporting fault-tolerant parallel programming in linda,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, pp. 287–302, March 1995. [Online]. Available: <http://portal.acm.org/citation.cfm?id=203121.203132>
- [33] Mozilla.org, “SpikerMonkey (JavaScript-C) Engine,” <http://www.mozilla.org/js/spidermonkey/>. [Online]. Available: <http://www.mozilla.org/js/spidermonkey/>
- [34] Python.org, “Python Programming Language - Official Website,” <http://python.org/>. [Online]. Available: <http://python.org/>
- [35] Lua.org, “The Programming Language Lua,” <http://www.lua.org/>. [Online]. Available: <http://www.lua.org/>
- [36] ECL, “ECL - A Common-Lisp Implementation,” <http://ecls.sourceforge.net/>. [Online]. Available: <http://ecls.sourceforge.net/>
- [37] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible markup language (XML) 1.0,” *W3C recommendation*, vol. 6, 2000.
- [38] B. Pittel, “On spreading a rumor,” *SIAM J. Appl. Math.*, vol. 47, pp. 213–223, March 1987. [Online]. Available: <http://portal.acm.org/citation.cfm?id=37387.37400>
- [39] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *Computational Science Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, Jan.–Mar. 1998.
- [40] V. S. Sunderam, “PVM: A framework for parallel distributed computing,” *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315–339, 1990.
- [41] “Entangled: DHT and tuple space based on Kademia.” [Online]. Available: <http://entangled.sourceforge.net/>

On the Calculation of the Checkpoint Interval in Run-Time for Parallel Applications

Leonardo Fialho*, Dolores Rexachs and Emilio Luque

Department of Computer Architecture and Operating System, University Autònoma of Barcelona, Spain
leonardo.fialho@caos.uab.es, dolores.rexachs@uab.es, emilio.luque@uab.es

Abstract—*The growth in the number of components that compose parallel computers increases their fault frequency. Currently, in such systems faults are no longer a rare event but a common problem, thus some sort of fault tolerance should be provided. In general, fault tolerance protocols rely on checkpoints. A common question surrounding checkpointing is the definition of the checkpoint interval. Checkpoint interval models define variables which depends on application characteristics, e.g. the time need to take a checkpoint. The use of average values and/or statistical data to define these variables reduces the model's accuracy. In this paper we propose a methodology to define in run-time the variables value needed to calculate the checkpoint interval. While using uncoordinated checkpoint this interval can be defined individually for each process of the parallel application. The variables definition relies on the measuring of the time spent on fault tolerance tasks in run-time. Experimental evaluation shows that the use of our methodology reduces in more than 3% the overhead introduced by fault tolerance while tested applications are running in a faulty environment.*

Keywords: MPI; fault tolerance configuration; checkpoint interval; uncoordinated checkpoint.

1. Introduction

The growth in the number of components that compose parallel computers increases is notorious for increasing their fault frequency [1]. Currently, in such systems faults are no longer rare events but a common problem. Some systems such as the BlueGene/L the Mean Time Between Failures (MTBF) is counted in days. However, the commodity clusters exhibit a usual MTBF of tens of hours [2]. The natural answer for this problem is to provide some sort of fault tolerance for applications running on these systems. This permits applications to finish successfully despite faults.

To write applications with native support for faults seems to be a good option. There are many techniques that help developers to codify fault tolerant parallel applications [3]. Many of these techniques are suitable to be used in conjunction with MPI: a widely used message passing library

for parallel programming. However, this approach requires the rewriting of legacy applications. Another solution is to provide fault tolerance at the communication library level and on the parallel environment. The combination of a resilient parallel environment and a fault recovery technique had been useful in MPI implementations like MPICH [4] and Open MPI [5][6].

To save the application state and to resume its execution in case of faults is commonly known as rollback-recovery. There are different rollback-recovery protocols that can be useful to assure application completion [7][8]. These protocols ultimately rely on checkpoints as the main state-saving technique or to save storage space while using combined with message logging *i.e.* reducing the space needed to store message log. The matter in question which surrounds checkpointing is the definition of the frequency in which checkpoints should be taken, better know as the checkpoint interval. If the checkpoint interval is smaller or bigger than the optimal the overhead added by the fault tolerance increases [9]. Because checkpointing is a widely used technique, there have been studies regarding the definition of its interval since the 70's [10] until today [11].

However, these studies are far from being the ultimate solution to the checkpoint interval. The major root cause of this resides in the definition of the variables value used by these models. The use of average values as input parameters for models reduces their accuracy. During the execution, some application characteristics may change over the time. Thus, models will experience a loss of accuracy because the checkpoint interval does not change to reflect such changes. Models variables depend on the application characteristics such as the memory footprint and the communication pattern, besides the system load such as the storage and the communication network.

In this paper we propose a methodology to define in run-time the checkpoint interval for parallel applications. The dynamic definition relies on the measuring of the time spent on fault tolerance tasks to obtain values for the checkpoint interval model variables. It turns the checkpoint interval model versatile enough to accommodate changes in the application characteristics throughout its execution. Experimental evaluation shows that the use of our methodology reduces in more than 3% the overhead introduced by fault tolerance while tested applications are running in a faulty environment.

This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974.

*Contact author to whom correspondence should be addressed.

†This paper is addressed to the PDPTA conference.

The content of this paper is organised as follows. The related work is introduced in section 2. In section 3 a description of the model used and some improvements made on it is presented. Results are shown in section 4. Conclusions are stated in section 5 besides future work.

2. Related Work

In the last years many fault tolerance MPI implementations has been designed. In general, those implementations rely on a rollback-recovery protocol. Such protocols are based on checkpointing, message logging, or both combined [7]. The era in which parallel applications are unable to finish due to faults in the parallel machine has gone.

However, due to the overhead introduced by fault tolerance tasks, especially by the recovery phase, researchers start to work on adaptive fault tolerance [12][13][14]. Adaptive fault tolerance requires information such as status and error reports about the machine in which applications are running. Indeed, a framework has been designed [15] to provide such information to fault tolerance libraries. This permits the creation of runtime strategies to dynamically reconfigure the parallel environment to avoid application being affected by faults [16].

Furthermore, there is a lack of studies regarding the configuration of the fault tolerant strategy according to specific applications characteristics. Working on this direction, Chen and Ren have published a study about the impact of the checkpoint interval on soft real-time applications [14]. Moreover, recently Jones et al. have published a work about the impact of a misconfigured checkpoint interval on the application efficient [9]. Despite of this, there are too few studies about the dynamic definition of the fault tolerance configuration according to specific application requirements.

3. The Methodology to Define the Checkpoint Interval in Run-Time

Our propose to define the checkpoint interval in run-time rests on two foundations: first on a checkpoint interval model and second on the measurement of the time needed to perform fault tolerant tasks. The second provides the values for the variables used by the first.

To help familiarise the reader with the checkpoint interval model used in this paper, the following list provides useful definitions:

- α as the mean time to interrupt (MTTI) for a given system, which is the inverse of the fault probability.
- σ as the checkpoint interval used to run the application.
- t_c as the time spent on a checkpoint operation including the storage time.
- t_l as the time needed to load a checkpoint from storage, not the rework time.

- Δ_{lp} as the time added to message delivery due to the logging procedure.
- Δ_{lr} as the time spent on processing the message log after a fault.
- ϕ as the factor which represents the inter-process dependency [17].

To define the value of these variables, we propose a monitoring mechanism of the fault tolerant tasks performed during application execution. The diagram shown in figure 1 depicts such a mechanism.

The time needed to perform a checkpoint operation (t_c) is measured by the timer depicted in events 1 and 2 of the diagram. The inter-process dependency factor is calculated by analysing sources and destinations of messages exchanged with other processes and is depicted in the diagram by event

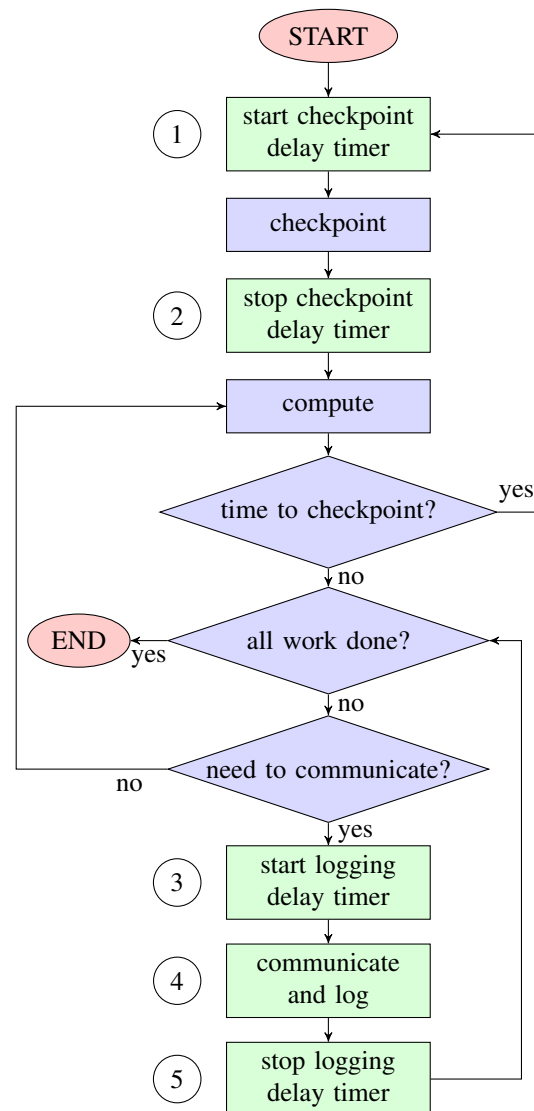


Fig. 1: Diagram of the methodology used to define model variables values in run-time.

4. The message logging overhead depends on the logging protocol used [7].

In this paper we will analyse the overhead introduced by a pessimistic receiver-based message logging. The time added to message delivery due to the logging procedure (Δ_{lp}) is measured by the timer depicted in events 3 and 4. When the message logging operation is performed on the receiver process, as shown in figure 2, in case of faults, only the faulty process is involved in recovery. Since messages do not need to be replayed, the time needed to process the message log (Δ_{lr}) tends to be unappreciable. Thus, the value of the Δ_{lr} variable can be considered as zero [19].

The time needed to load a checkpoint (t_l) cannot be measured using our methodology if no fault occurs. However, as a first approximation we consider this time equal to the time needed to perform a checkpoint. It does not reduce the checkpoint interval model accuracy because variables related to the recovery phase, with the exception of the rework time, tend to be inappreciable [10].

After all variables values needed by the checkpoint interval model had been already defined in run-time it is possible to use the following checkpoint interval model to calculate the checkpoint interval.

For a system with a known MTTI, the following equation^[1] estimates the wall-clock time T_{est} required to run an application (which originally takes T_p time to conclude) in a faulty environment with fault tolerance:

$$T_{est} = T_p \left[1 + \frac{\phi\sigma^2 + \sigma(2\phi t_l + \phi t_c + 2\phi\Delta_{lr} - t_c + 2\Delta_{lp})}{\alpha(2\sigma + 2t_c)} + \frac{2t_c(\phi t_l + \phi\Delta_{lr} + \alpha - t_l - \Delta_{lr} + \Delta_{lp})}{\alpha(2\sigma + 2t_c)} \right] \quad (1)$$

¹The definition of the checkpoint interval model used in this paper can be found in http://caos.uab.es/~lfialho/ic/parallel_model.pdf.

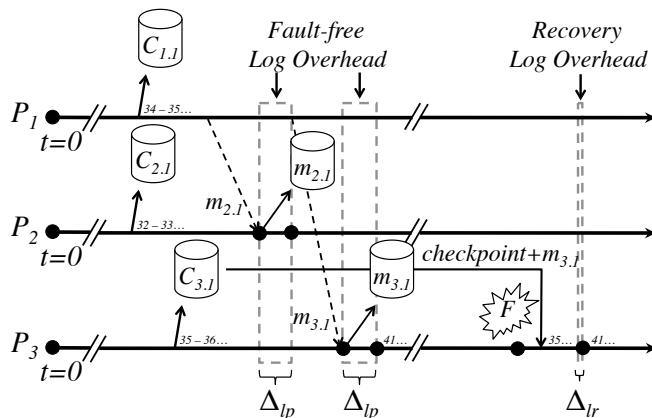


Fig. 2: Overhead introduced by a pessimistic receiver-based message logging protocol during a fault-free execution (Δ_{lp}) and during the recovery phase (Δ_{lr}).

In the equation above, the fault detection latency has been omitted. This variable can be safely omitted because it depends on the fault detection mechanism implemented by the fault tolerance architecture. If we suppose the use of a heartbeat/watchdog system, then the higher the heartbeat frequency, the smaller the detection time. Moreover, the heartbeat communication does not impose a considerable overhead on the system. In addition, there is always the possibility of using the application communication as another fault detection mechanism [18].

The checkpoint interval that minimises the fault tolerance overhead of the aforementioned model is:

$$\sigma = \frac{\sqrt{\phi t_c(t_c + 2\alpha - 2t_l - 2\Delta_{lr})}}{\phi} - t_c \quad (2)$$

and the inter-process dependency factor is defined by the following equation:

$$\phi_{global} = \frac{\sum_1^N P(n)}{N^2} \quad (3)$$

where $P(n)$ is the function which defines the number of processes that depend on the process n including itself. N is the total number of processes in the parallel application.

There is an issue with the whole model presented related to the definition of the inter-process dependency factor. Equation 3 reflects this inter-process dependency factor for the entire application. This factor should be redefined to represent the dependency of an individual process in relation to other processes. Below we present our propose to the redefinition of this factor:

$$\phi = \frac{P(n)}{N} \quad (4)$$

and finally, the value for this factor is defined in run-time based on the application monitoring. As in previous equation, $P(n)$ is the function which defines the number of processes that have sent to or received messages from process n , including itself. N is the total number of processes in the parallel application.

Our methodology is based on measurements taken during the most recently checkpoint cycle. When the application changes its behaviour, *i.e.* the communication pattern, or its memory footprint, after one checkpoint cycle the checkpoint interval will already be adapted to the new application characteristics. Moreover, except during the start-up and finalisation it is expected that applications do not change their behaviour or memory footprint too frequently in comparison to the checkpoint interval [20].

4. Experimental Evaluation

To evaluate our proposal, three experiments were designed. In the first experiment we depict the adaptation of the checkpoint interval to changes in the process size and we

also demonstrate the influence of the communication pattern on the inter-process dependency factor (ϕ). The second experiment depicts the reduction in the overhead introduced by fault tolerance while using our methodology to define the checkpoint interval in run-time. Finally, the third experiment shows the accuracy of the inter-process dependency factor defined in run-time.

Experiments run in an 8-node cluster. Each node was equipped with two Dual-Core Intel Xeon processors running at 2.66GHz, 12 GBytes of main memory, and a 160 GBytes SATA disk for local storage. Nodes were interconnected via two Gigabit Ethernet interfaces. One of these networks was used for storage while the other was used for process communication. RADIC/OMPI [6] was used as a fault tolerant MPI library.

To create a fault scenario processes are killed during the application execution. The fault moment is defined by the MTTI (α) and its distribution along the MTTI is defined using the MT19937 PRNG algorithm [22]. The processes to be killed is selected using the same algorithm. After the fault has been injected the node is available to be reused by a recovered process.

4.1 Checkpoint Interval Adaptation

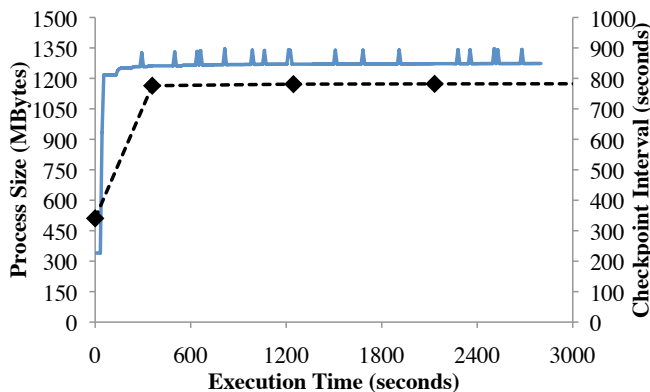
Model variables such as t_c and t_l depends on the amount of memory used by application processes. And processes on the same application may present different memory footprints. This occurs because processes compute different data or processes play different roles in the parallel application.

To depict the adaptation of the checkpoint interval to the process memory footprint we have used the NAMD molecular dynamics application [21]. NAMD is implemented over a Master/Worker paradigm where workers also communicate between themselves; the master process requires more memory in comparison to the workers. The experiment has been executed using with a fault frequency (α) of 3600 seconds and the heartbeat frequency (t_d) was set to 1 second. Values for t_c , t_l , Δt_p , and Δt_r were measured during the execution. For this application we have manually calculated the inter-process dependency factor (ϕ) and its value is 1.

Dashed lines in figures 3 depict the checkpoint interval used throughout the application execution. Figure 3(a) refers to the master process, while figure 3(b) refers to a worker process. Figure 3 depicts only one worker processes, however others present a similar behaviour.

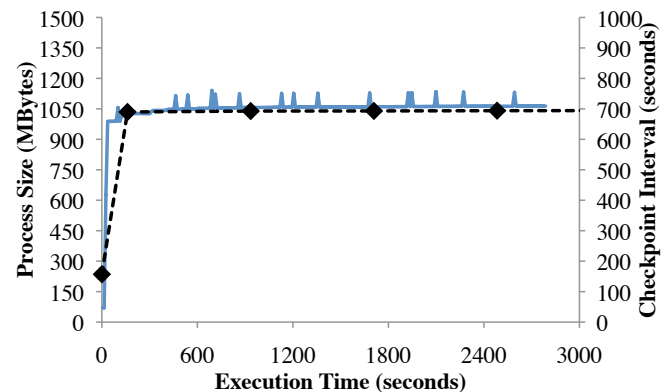
As the figures depict, processes use a small amount of memory in the startup phase. As a consequence of this the model calculates a short checkpoint interval initially. However, after the startup phase the application increases its memory footprint. After the second checkpoint the checkpoint interval changes to reflect the changes on the process memory footprint. Tables in figure 3(a) and 3(b) summarise the checkpoint instances and sizes for the master and a worker process, respectively.

To depict the adaptation of the checkpoint to the inter-process dependency factor we have used a dynamic matrix multiplication application built under a Master/Worker paradigm where workers only communicate with the master



Execution Instant	Process Size	Checkpoint Interval
0.31 seconds	339.96 MB	340.56 seconds
358.45 seconds	1261.75 MB	775.93 seconds
1245.35 seconds	1270.05 MB	779.04 seconds
2132.25 seconds	1272.37 MB	779.41 seconds

(a)



Execution Instant	Process Size	Checkpoint Interval
0.19 seconds	70.57 MB	157.35 seconds
160.95 seconds	1026.79 MB	689.77 seconds
934.97 seconds	1056.99 MB	693.29 seconds
1708.81 seconds	1060.23 MB	693.87 seconds

(b)

Fig. 3: The continuous line shows the memory footprint of the NAMD (a) master and (b) worker processes running the “stmv” workload; values are shown on the left axes. The dashed line represents the checkpoint interval used; values are shown on the right axes. The rhombus points depict checkpoint instances. Tables depict first four values of the checkpoint size and the calculated next checkpoint interval for each type of process according to the execution instant.

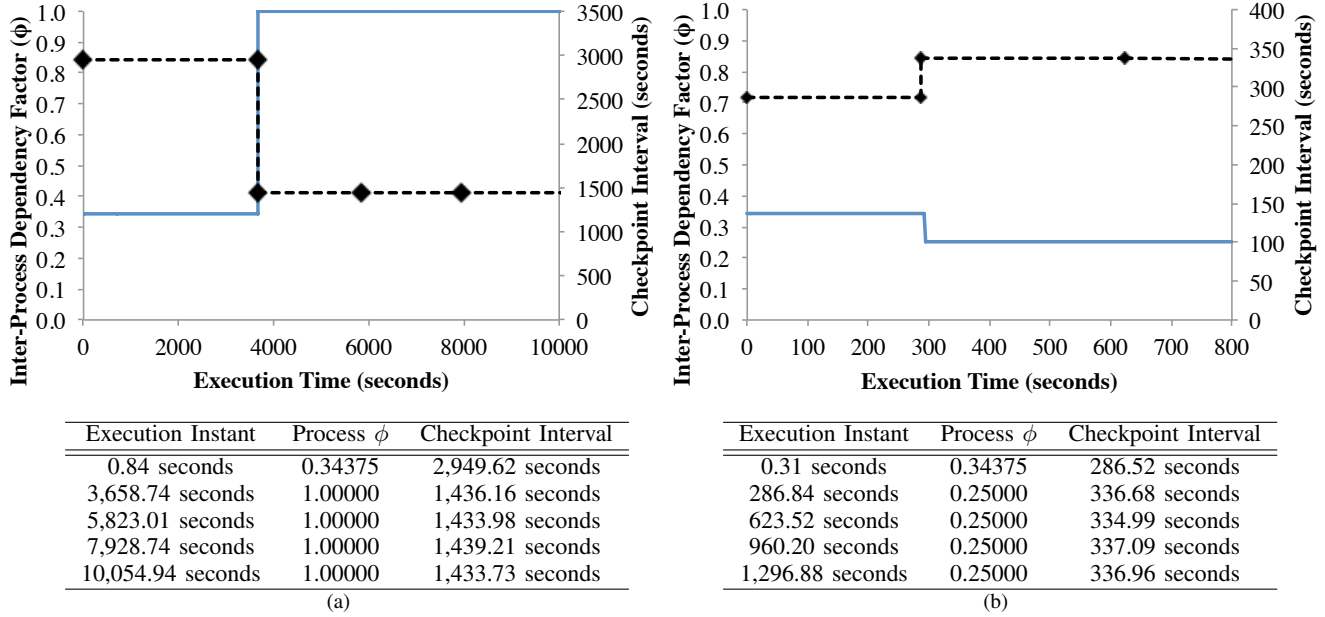


Fig. 4: The continuous line shows the value of ϕ for the matrix multiplication (a) master and (b) process; values are shown on the left axes. The dashed line represents the checkpoint interval used; values are shown on the right axes. The rhombus points depict checkpoint instances. Tables depict first five values of ϕ and the calculated next checkpoint interval for each type of process of the matrix multiplication execution according to the execution instance.

process. This parallel application was executed using 8 nodes.

Considering the equation 3, the initial value for the ϕ variable is 0.34375. This value represents a global view of the relationship established between all processes on this parallel application. The fault frequency (α) has been defined as 3600 seconds and the heartbeat frequency (t_d) has been set to 1 second. Values for t_c , t_l , Δ_{lp} , Δ_{lr} , and ϕ are measured during the execution. Equation 4 has been used to define in run-time the value of ϕ for each process.

Continuous lines in figure 4 depict the calculated values for the ϕ in run-time for the master 4(a) and for a worker 4(b) process, respectively. In addition, the dashed line on these figures depict the values of the checkpoint interval during the application execution as well as the checkpoints instances.

As shown in figure 4(a), the initial value of 0.34375 was redefined to 1. This occurs because between the first and the second checkpoint the master process communicated with all 7 workers. As a consequence of this increase in the value of ϕ , the model has changed the checkpoint interval. Similarly, in figure 4(b) the decrease in the value of ϕ increases the time between checkpoints for a worker process.

Figure 4 depicts only one worker process, however, other worker processes present similar behaviour. The huge difference between the checkpoint interval calculated for the master and for the worker process is caused by the difference in the memory footprint of these processes.

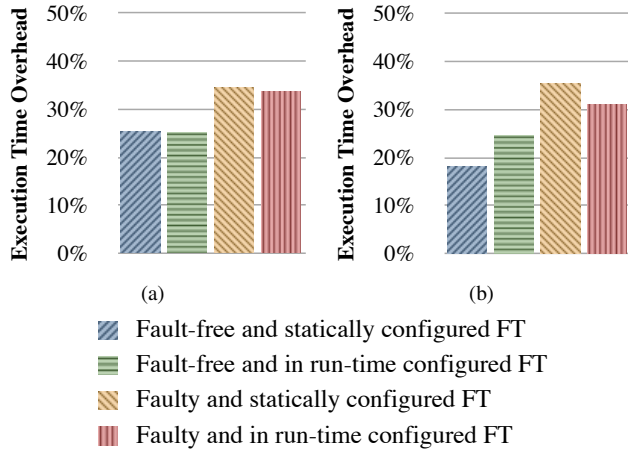
4.2 Reduction in the Overhead

The next experiments depict the performance gain in using our methodology to define the checkpoint interval in run-time. This experiment compares the performance of our proposal with a static configuration in a faulty and fault-free scenario. The comparison was made using the aforementioned NAMD and dynamic matrix multiplication applications. In this experiment only one fault was injected in each execution. The moment of the fault differs from one execution to other. The fault is distributed along the application execution according to the MT19937 PRNG algorithm. Each experiment has been executed at least in 16 times and values are the average of all data that fall in a 95% confidence interval.

As shown in figure 5(a) the use of the fault tolerance provided by the RADIC/OMPI library introduces an overhead of about 25% in a fault-free execution and about 34% in a faulty scenario.

As shown in table in figure 5 there is a modest reduction in the overhead while the checkpoint interval is calculated in run-time. This occurs because there is no significant change in the NAMD processes characteristics, except for the memory footprint in the start-up phase. However, as shown in figure 5(b) the matrix multiplication application presents different results.

In the fault-free environment the execution using statically configured checkpoint interval presents a smaller overhead than the execution running with in run-time configuration.



App	Statically Configured		Configured in Run-Time	
	Fault-free	Faulty	Fault-free	Faulty
NAMD	25.4%	25.1%	34.5%	33.6%
MM	18.1%	24.6%	35.4%	31.1%

Fig. 5: Comparison of the (a) NAMD and (b) a matrix multiplication execution time using different fault tolerance configuration strategies on different environments.

This is because the initial global value of ϕ increases the checkpoint interval for the master process. This reduces the number of checkpoints performed. In this situation, the overhead introduced by a fault increases. This can be verified when we compare the total wall time clock in a faulty environment for the configurations made statically and in run-time.

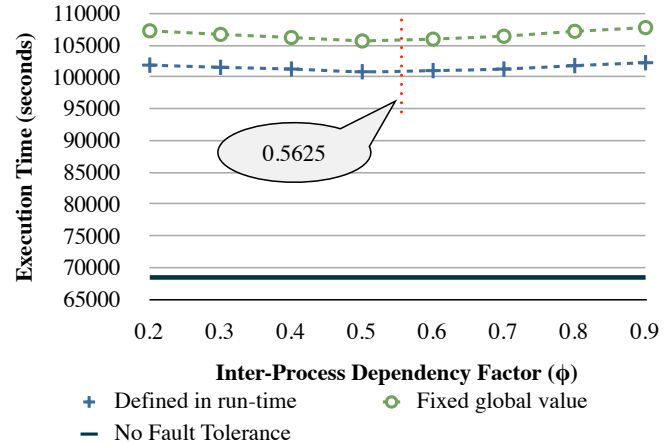
4.3 The Accuracy of the Inter-Process Dependency Factor Defined in Run-Time

To verify the accuracy of our model we executed the NAS [23] LU class B with 8 processes modified to iterate 300,000 times. This modified version of the LU has been executed using different global values for the ϕ , from 0.9 to 0.2. Table 1 shows the correct value of this factor individually and globally for this application.

Analysing the curve in figure 6 and the data present in the table it is possible to guess that the optimum value for the ϕ for this execution should be a value between 0.45 and 0.60. Despite of the small difference between the global and the individualised values for the ϕ , to use of a precise value for this factor reduces in more than 3% the overhead introduced

Table 1: Values for the inter-process dependency factor for the entire LU application and for each process individually.

Process Rank (Global Value)	$P(n)$	ϕ
Running with 8 processes		0.56250
0, 3, 4, 7	4	0.50000
1, 2, 5, 6	5	0.62500



ϕ	Overhead Using Fixed Global Value	Overhead Using a Defined in Run-Time Value	Difference
	0.2	36.2%	
0.3	35.8%	32.6%	3.27%
0.4	35.5%	32.4%	3.16%
0.5	35.2%	32.1%	3.11%
0.6	35.4%	32.2%	3.17%
0.7	35.7%	32.4%	3.28%
0.8	36.1%	32.7%	3.39%
0.9	36.5%	33.0%	3.43%

Fig. 6: Comparison between static and in run-time configured values of the inter-process dependency factor.

by the fault tolerance tasks for this application.

5. Conclusions

Checkpoint interval models used to rely on input variables based on average values. This reasoning is valid for applications running for a long time on systems that present a high fault frequency. However, this is not the common environment faced by parallel application users. This paper has presented a methodology to dynamically define the input variables used by models based on measurements performed during the application execution.

We propose the monitoring of processes that compose the parallel application to achieve the values for the variables used by checkpoint interval model. We monitor the time needed to perform fault tolerant tasks as well as the number of peers each process communicates with.

This instrumentation allow the definition of the checkpoint interval in run-time with a high degree of precision, process by process. The use of this methodology reduces in about 3% the overhead introduced in the execution time for applications running in faulty environments.

5.1 Future Work

The overhead added to the application execution by the monitoring mechanism tends to be unappreciable. However,

it is necessary to quantify this overhead.

The use of uncoordinated checkpointing is the only solution that allows the use of different checkpoint intervals for each application process. However, the use of uncoordinated checkpointing assisted by message logging may not be the solution that presents the lowest overhead. There is the need to analyse if a sender-based message logging or a coordinated checkpointing solution present better results.

References

- [1] F. Cappello, "Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities," *International Journal of High Performance Computing Applications*, pp. 212—226, 2009. [Online]. Available: <http://dx.doi.org/10.1177/1094342009106189>
- [2] A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the Message Logging Model for High Performance," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2196—2211, 2010. [Online]. Available: <http://dx.doi.org/10.1002/cpe.1589>
- [3] W. Gropp and E. Lusk, "Fault Tolerance in Message Passing Interface Programs," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363—372, 2004. [Online]. Available: <http://dx.doi.org/10.1177/1094342004046045>
- [4] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging," *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, p. 25, 2003. [Online]. Available: <http://dx.doi.org/10.1109/SC.2003.10027>
- [5] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI," *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*, 2007. [Online]. Available: <http://dx.doi.org/10.1109/IPDPS.2007.370605>
- [6] L. Fialho, G. Santos, A. Duarte, D. Rexachs, and E. Luque, "Challenges and Issues of the Integration of RADIC into Open MPI," *Proceedings of the 16th European PVM/MPI Users' Group Meeting*, pp. 73—83, 2009. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1612227>
- [7] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375—408, 2002. [Online]. Available: <http://dx.doi.org/10.1145/568522.568525>
- [8] S. Kalaiselvi and V. Rajaraman, "A survey of checkpointing algorithms for parallel and distributed computers," *Sadhana*, vol. 25, no. 5, pp. 489—510, 2000. [Online]. Available: <http://dx.doi.org/10.1007/BF02703630>
- [9] W. Jones, J. Daly, and N. DeBardleben, "Impact of Sub-optimal Checkpoint Intervals on Application Efficiency in Computational Clusters," *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 276—279, 2010. [Online]. Available: <http://dx.doi.org/10.1145/1851476.1851509>
- [10] J. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530—531, 1974. [Online]. Available: <http://dx.doi.org/10.1145/361147.361115>
- [11] J. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303—312, 2006. [Online]. Available: <http://dx.doi.org/10.1016/j.future.2004.11.016>
- [12] Y. Li and Z. Lan, "Exploit Failure Prediction for Adaptive Fault-Tolerance in Cluster Computing," *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid*, pp. 531—538, 2006. [Online]. Available: <http://dx.doi.org/10.1109/CCGRID.2006.45>
- [13] Z. Lan, Y. Li, Z. Zheng, and P. Gujrati, "Enhancing Application Robustness through Adaptive Fault Tolerance," *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing*, 2008. [Online]. Available: <http://dx.doi.org/10.1109/IPDPS.2008.4536383>
- [14] N. Chen and S. Ren, "Adaptive Optimal Checkpoint Interval and Its Impact on System's Overall Quality in Soft Real-time Applications," *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 1015—1020, 2009. [Online]. Available: <http://dx.doi.org/10.1145/1529282.1529506>
- [15] R. Gupta, P. Beckman, B. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra, "CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems," *Proceedings of the 2009 International Conference on Parallel Processing*, pp. 237—245, 2009. [Online]. Available: <http://dx.doi.org/10.1109/ICPP.2009.20>
- [16] Y. Li, Z. Lan, P. Gujrati, and X. Sun, "Fault-Aware Runtime Strategies for High Performance Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 4, pp. 460—473, 2009. [Online]. Available: <http://dx.doi.org/10.1109/TPDS.2008.128>
- [17] L. Fialho, D. Rexachs, and E. Luque, "What Is Missing in Current Checkpoint Interval Models?" **To appear in the Proceedings of the 31th International Conference on Distributed Computing Systems**, 2011.
- [18] A. Duarte, D. Rexachs, and E. Luque, "Increasing the cluster availability using RADIC," *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, 2006. [Online]. Available: <http://dx.doi.org/10.1109/CLUSTER.2006.311872>
- [19] S. Rao, L. Alvisi, and H. Vin, "The Cost of Recovery in Message Logging Protocols," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 160—173, 2000. [Online]. Available: <http://dx.doi.org/10.1109/69.842260>
- [20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 45—57, 2002. [Online]. Available: <http://dx.doi.org/10.1145/605397.605403>
- [21] J. C. Philips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable Molecular Dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781—1802, 2005. [Online]. Available: <http://dx.doi.org/10.1002/jcc.20289>
- [22] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3—30, 1998. [Online]. Available: <http://dx.doi.org/10.1145/272991.272995>
- [23] W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow, "New Implementations and Results for the NAS Parallel Benchmarks 2," *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.3199>

Defining the Checkpoint Interval for Uncoordinated Checkpointing Protocols

Leonardo Fialho*, Dolores Rexachs and Emilio Luque

Department of Computer Architecture and Operating System, University Autònoma of Barcelona, Spain
leonardo.fialho@caos.uab.es, dolores.rexachs@uab.es, emilio.luque@uab.es

Abstract—*Parallel applications running on large computers suffer from the absence of a reliable environment. Fault tolerance proposals, in general, rely on rollback-recovery strategies supported by checkpoint and/or message logging. There are well-defined models that address the optimum checkpoint interval for coordinated checkpointing. Nevertheless, there is a lack of models concerning uncoordinated checkpointing combined with message logging. First we present a model designed for serial applications or coordinated checkpointing-based solutions. Our contribution is the extension of this model to a scenario based on uncoordinated checkpointing combined with message logging. We introduce two key points to minimise the fault tolerance overhead for parallel applications. The first is the use of a factor to represent the dependency relation between processes. The second is the use of a specific checkpoint intervals for each process. Experiments show that our model performs as well as previous studies for serial applications or coordinated checkpointing. While running parallel applications using uncoordinated checkpointing combined with message logging, our checkpoint interval model effectively minimises the overhead introduced by the fault tolerance tasks. Moreover, the overhead prediction error is smaller than 5% for all applications tested.*

Keywords: MPI; fault tolerance; checkpoint interval; model; uncoordinated checkpoint.

1. Introduction

Fault tolerance has become an important issue for parallel applications in the last few years. The growth of the number of components, which form parallel machines, are the major root causes of the failures increasingly seen on these machines. In order to achieve the execution end, parallel applications should use some fault tolerance strategy. Strategies can be the use of redundant hardware or the incorporation of the redundancy by software. Actually, the second is cheaper than the first even though it represents an overhead on the application run time.

This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974.

*Contact author to whom correspondence should be addressed.

†This paper is addressed to the PDPTA conference.

Checkpointing is an established rollback-recovery technique used to achieve fault tolerance on applications. There are well-defined models [1], [2] to calculate the checkpoint interval to minimise the overhead introduced by fault tolerance and maximise the application efficiency, *i.e.*, “the ratio of time the job spends making forward progress compared to the entire wall-clock time” [3]. Nevertheless, these models have been designed based on serial applications. The use of models based on serial applications is acceptable for parallel applications when they are protected by coordinated checkpointing. However, for uncoordinated checkpointing-based strategies these models would not be useful.

Fault tolerance architectures designed for parallel machines such as MPICH-V [4] and RADIC [5] are based on uncoordinated checkpointing combined with message logging. Ergo, these architectures suffer from the lack of models to calculate the checkpoint interval as well as to predict the overhead introduced by the fault tolerance architecture on the application run time.

In this study we will propose a novel model to calculate a checkpoint interval to minimise the overhead introduced by fault tolerance architectures based on uncoordinated checkpointing combined with message logging in parallel applications. For the sake of deducing the parallel application fault tolerance model, a model based on serial applications will be introduced first. The model to calculate the checkpoint interval for parallel application incorporates the message logging influence on the overhead and a factor to measure the dependency relationship between processes. We present two key points to minimise the fault tolerance overhead in parallel applications. The first is the use of a factor to represent the dependency relationship between parallel application processes. The second is the use of different checkpoint intervals for each parallel application process based on its own characteristics.

Experiments show that our model performs as well as previous studies while running serial applications. For parallel applications the overhead prediction error is less than 5% while running with uncoordinated checkpointing combined with message logging.

The content of this paper is organised as follows. The related work is presented in section 2. Section 3 introduces the checkpoint interval models. The experimental evaluation comes in section 4. The conclusions are stated in section 5.

2. Related Work

The use of analytical models to define the optimum checkpoint interval for serial applications has been studied from the 70's until today. In 1974, Young [6] introduced the first order approximation, an analytical model to determine the optimum checkpoint interval. Using Young's model it is possible to calculate the optimum checkpoint interval once the user knows the time needed to perform a checkpointing operation and the system fault probability. In order to predict the overall running time while performing checkpoints, Young's model requires the original application run time as well as the time needed to recover a failed process. More recently, Lin *et al.* [7] and Gropp and Lusk [1] presented a model achieving the same result for the optimum checkpoint interval. Nevertheless, they use different approaches that lead to different overhead prediction equations.

Many years after Young, Daly presented a deeply analytical study [2] to determine the higher order estimation of the optimum checkpoint interval. Daly analyses different scenarios such as multiple failures between checkpoints, fractional rework (the amount of work completed after a checkpoint and prior to a failure), failures during restarts, *et cetera*.

Despite Gelenbe *et al.* [8] and Ken and Mark's [9], whom were analysing models for distributed computing, there is a lack of knowledge surrounding the optimum checkpoint interval for parallel applications. Currently, many studies about fault tolerance for parallel applications are limited to presenting the approach used to achieve protection. Most of these studies discuss differences between coordinated and uncoordinated checkpointing, logging strategies, implementation details, architecture design, *et cetera*. In these studies in general, authors have omitted the method used to calculate the checkpoint interval used in experiments. A rare exception is Bouteiller *et al.* [10] who describe the model employed to calculate the checkpoint interval used to run experiments to depict the impact of fault frequency on the application run time. Nevertheless, this model is not useful to define the checkpoint interval to minimise the fault tolerance overhead.

Models designed to be used on serial applications can also be used on parallel applications under certain circumstances (*e.g.* if the application is protected by coordinated checkpointing). However, after a fault the system restores the last checkpoint and work done after the checkpoint and before the fault is lost.

To understand the impact of faults on parallel applications let us suppose a parallel application running with a rollback-recovery fault tolerance assisted by coordinated checkpointing. Checkpointing and recovery are collective operations that involve all processes in the parallel application. Thus, all processes are checkpointed at the same time and after a fault all processes should roll back and resume their operations from the last checkpoint. In the case of coordinated checkpointing models designed for serial applications can

also be used.

However, parallel applications running with a fault tolerance system which implements an uncoordinated checkpointing protocol cannot use existing models to calculate the optimum checkpoint interval. In this scenario, after the fault occurrence only the faulty process needs to be rolled back. In this case just the work done by the faulty process is lost. Other processes continue to compute.

As far as we know, there is no model to calculate the checkpoint interval to minimise the overhead introduced by fault tolerance in this case. Also, there is no model to predict the overhead introduced by fault tolerance on parallel applications running with uncoordinated checkpointing.

3. Developing the Checkpoint Interval Models

In order to better understand the deduction of the checkpoint interval models let us consider figure 1 and the following naming system:

- t_c as the time spent on a checkpoint operation including the storage time. In other words, it is the application interruption time necessary to take a checkpoint.
- t_d as the time needed to detect a fault, also known as fault detection latency.
- t_l as the time needed to load a checkpoint from storage.
- t_r as the amount of time needed to recover a failed process and achieve the computation point just before fault. It is the reworking of the previous lost computation, also known as fractional rework.
- Q as the quantity of checkpoints that should be performed between two faults.
- T_c as the total protection time represented by the sum of all t_c between two faults. This value can be obtained multiplying t_c by Q . If there is some overhead introduced by the logging procedure this time need to be take in account.
- T_r as the total recovery time per fault represented by the sum of t_d , t_l , and t_r .
- α as the mean time to interrupt (MTTI) for a given system, which is the inverse of the fault probability.
- σ as the checkpoint interval used to run the application. It can also be considered as the useful time for the application to compute.
- Δ_{lp} as the time added to message delivery due to the logging procedure, if it exists.
- Δ_{lr} as the time spent no processing the message log after a fault. The majority of this is the replaying time, if it exists.

As shown in figure 1, the recovery task occurs at the beginning of the period between faults F_x and F_y . Recovery takes T_r time (segment \overline{BE}) to conclude and after this

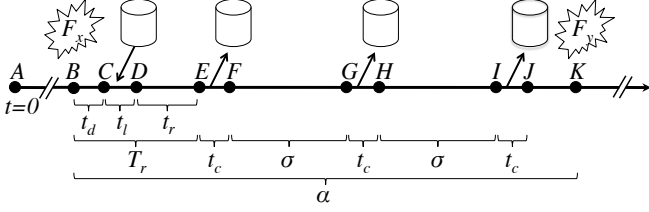


Fig. 1: Between faults F_x and F_y there is a recovery time T_r and 3 checkpoints t_c among computational periods σ .

comes one or more computational segments followed by checkpoints. Each checkpoint requires t_c time (segments \overline{EF} , \overline{GH} , and \overline{IJ}) to be taken. Checkpoints are separated by an application computational period represented by σ (segments \overline{FG} and \overline{HI}). Ergo, σ is the interval between checkpoints, *ipso facto* the checkpoint interval. Segment \overline{JK} will be lost due to fault F_y . This work will be redone after the next recovery phase (not depicted in figure 1). As aforementioned, T_c is the sum of all t_c , it represents the sum of segments \overline{EF} , \overline{GH} , and \overline{IJ} in figure 1. Time spent on protection and recovery tasks is not useful application time, thus these tasks are considered to be overhead. This assumption leads us to the following equation:

$$\text{Overhead} = T_r + T_c \quad (1)$$

3.1 The Model for Serial Applications

As mentioned previously, T_r is the sum of the fault detection latency (t_d), checkpoint loading from storage (t_l) and the fractional rework (t_r). It can also be seen in figure 1. As proved by Daly [2], it is accepted to assume interrupts occur halfway through the checkpoint interval. Thus, we consider the fractional rework (t_r) as half of the checkpoint interval (σ), which leads us to the following equation:

$$T_r = t_d + t_l + \sigma/2 \quad (2)$$

The same demonstration can be used in relation to the fault detection latency. However, there are many fault detection mechanisms that can be used. Because it is a user-defined variable we will leave this variable untouched.

To calculate T_c the number of checkpoints (Q) should be defined. Q represents the number of segments composed of checkpoint (t_c) and compute time (σ) that fits the period between faults (α), excluding the recovery time. It is reflected in the equation below:

$$T_c = Q * t_c \quad (3)$$

$$Q = (\alpha - T_r) / (\sigma + t_c) \quad (4)$$

Using equations 1, 2, 3, and 4 and applying some algebraic operations the following overhead equation is obtained:

$$\text{Overhead} = (\sigma^2 + 2(\sigma t_d + \sigma t_l + \alpha t_c)) / (2(\sigma + t_c)) \quad (5)$$

We can find the value of σ_{opt} that minimises the fault tolerance overhead by deriving the overhead equation 5 with respect to σ and setting the result to zero [1]. Considering the positive solution, this operation brings us to the following optimum checkpoint interval:

$$\sigma_{opt} = \sqrt{t_c^2 - 2t_c t_d - 2t_c t_l + 2\alpha t_c} - t_c \quad (6)$$

3.2 The Model for Parallel Applications

The message logging disturbance depends on the logging protocol used [11]. For the development of our model a pessimistic receiver-based message logging has been used. To better understand the deduction of the new model let us consider figure 2. This figure depicts the disturbances added by message logging operations and how failures impact on different application processes.

In general, receiver based logging doubles the time needed for message delivery because the data storage cannot be overlapped with message delivery as shown in figure 2. Modifying equations 2 and 3 to reflect the message logging overhead, leads us to the following equations for process n :

$$T_{r_n} = t_d + t_l + \sigma/2 + \Delta_{lr} \quad (7)$$

$$T_{c_n} = (Q * t_c) + \Delta_{lp} \quad (8)$$

In case of faults, only the faulty process needs to rollback and during its recovering phase there is no interaction with non-faulty processes [11]. However, once a process fails other processes that depend on the first could continue waiting for data from the first before continuing their execution. It means that processes have an intrinsic inter-dependent relationship. In this paper this relationship will be named *inter-processes dependency factor*. This factor affects the T_r lowering its weight on the overhead equation. Rewriting the

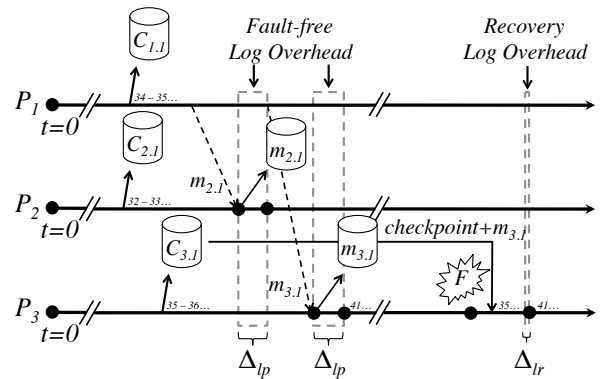


Fig. 2: Disturbances introduced by the receiver-based message logging protocol for protection Δ_{lp} and recovery Δ_{lr} .

overhead equation 1 to consider this assertion, the following equation is produced:

$$Overhead = \phi(T_r) + T_c \quad (9)$$

where ϕ represents the inter-process dependency factor. Regarding this factor, small values represent less dependency between processes while the higher value (1) means that when one process fails all other processes continue waiting for it. Replacing equations 7 and 8 in equation 9 and after applying some algebraic operations, the following equation is obtained:

$$Overhead = [\phi\sigma^2 + \sigma(2\phi t_d + 2\phi t_l + \phi t_c + 2\phi\Delta_{lr} - t_c + 2\Delta_{lp}) + 2t_c(\phi t_d + \phi t_l + \phi\Delta_{lr} + \alpha - t_d - t_l - \Delta_{lr} + \Delta_{lp})] / (2\sigma + 2t_c) \quad (10)$$

We can find the value of σ_{opt} that minimises the fault tolerance overhead for parallel applications by deriving equation 10 with respect to σ , and setting the result to zero[1]. Considering the positive solution, this operation brings us to the following optimum checkpoint interval:

$$\sigma_{opt} = \sqrt{\phi t_c(t_c + 2\alpha - 2t_d - 2t_l - 2\Delta_{lr})} / \phi - t_c \quad (11)$$

3.3 The Inter-Process Dependency Factor

All processes in the parallel application can fail. However, each process failure may impact other processes in a distinct way. This means that if process n fails, one or more processes can hang waiting for the recovery of n to be completed (*e.g.* considering a master/worker application, if the master process fails, all workers may wait for the recovery of the master). The following equation shows how to define the dependency factor using such analysis:

$$\phi = \frac{\sum_1^N P(n)}{N^2} \quad (12)$$

where $P(n)$ is the function that defines the number of processes that depend on the process n including itself, and N is the total number of processes in the parallel application.

Based on the example above, let us assume an application running with 8 processes and written under a master/worker paradigm in which workers do not communicate among themselves and supposing a function $P(n)$ which considers the existence of communication as the only dependency between processes. If the master process fails all workers should wait for its recovery, then $P(master)$ is 8. If any worker fails just the master may wait for it, then $P(worker)$ is 2 for all 7 workers. In compliance with this assumption the dependency factor for this application is 0.34375.

Besides message logging modelling, the introduction of the inter-process dependency factor is a key difference between the previous models and ours. The introduction of

this factor is crucial to the accuracy of the predicted checkpoint interval that minimises the fault tolerance overhead in parallel applications.

4. Experimental Evaluation

Hereunder, a comparison between models will be presented. The comparison was made using simulation and running real applications. The fault distribution is defined by the MTTI, and faults are displaced in time with a 100% of deviation calculated using the MT19937 PRNG algorithm [12]. Moreover, in this section we evaluate the checkpoint interval model for uncoordinated checkpointing.

To run experiments a 32 node cluster has been used. Each node is equipped with two Dual-Core Intel Xeon processors running at 2.66GHz, 12 GBytes of main memory and a 160 GBytes SATA disk for local storage. Nodes are interconnected via two Gigabit Ethernet interfaces. RADIC/OMPI [13] has been used as a fault tolerant MPI library.

To inject faults a program has been designed. This program runs on a machine external to the cluster. According to the fault distribution, the program connects to the target node and kills the application process. The target machine is selected in a round-robin fashion. The killed process is recovered from the last checkpoint by the daemon used to launch the application process.

4.1 Models for Serial Applications

To compare models we have used a simple matrix multiplication algorithm. This experiment tries to verify the accuracy of the calculated checkpoint interval to minimise the overhead introduced by fault tolerance. For that, the matrix multiplication has been executed with different checkpoint intervals. The application overhead is compared with the predicted overhead of the models. The fault detection latency is virtually zero and the MTTI (α) has been defined to be 100 seconds. Each experiment has been executed at least 16 times and values are the average of all data that fall in a 95% confidence interval.

Figure 3 depicts a comparison between overhead prediction of the models and a real execution. All models have calculated an optimum checkpoint interval between 9.75 and 10.3 seconds. All models have presented an overhead relative error smaller than 3%. Close to the optimum checkpoint interval the relative error is smaller than 2% for all models.

To evaluate the influence of the fault frequency a discrete event simulator has been used. Table 1 shows variables used to simulate a 500 days application as well as the values of the optimum checkpoint interval calculated by models for each scenario. The detection latency has been set to zero.

Figure 4(a) compares the simulated run time with the four models using a 24 hour MTTI. The results of the models are very close to the simulation. With regard to the predicted overhead, our model presents a relative error of 0.64% on values close to the calculated checkpoint interval (114.89

Table 1: Variables used to simulate the influence of MTTI on model accuracy and the optimum checkpoint interval.

MTTI	t_c	t_l	Fialho	Daly	Gropp	Young
hours	minutes	minutes	minutes	minutes	minutes	minutes
24	5	5	114.89	115.00	120.00	120.00
6	5	5	54.79	55.00	60.00	60.00

minutes). Daly's model presented an error smaller than 0.2% for all checkpoint intervals simulated.

Figure 4(b) demonstrates that Young's model cannot predict the overhead with less than 15% error when the MTTI is smaller than 6 hours and the time needed to take or load a checkpoint is 5 minutes. However, its calculated checkpoint interval is still close to the optimum. The smaller simulated run time is for a 55 minute checkpoint interval. At this point our model, Gropp's, and Daly's present a relative error of 2.54%, 1.23%, and 0.67%, respectively.

4.2 Analysing the Results of the Checkpoint Interval Model for Parallel Applications

To evaluate the results of the checkpoint interval model for parallel application two sets of experiments have been designed: 1) one analyses the effectiveness of the inter-process dependency factor, and 2) verifies the correctness of the message logging modelling. To show that the using

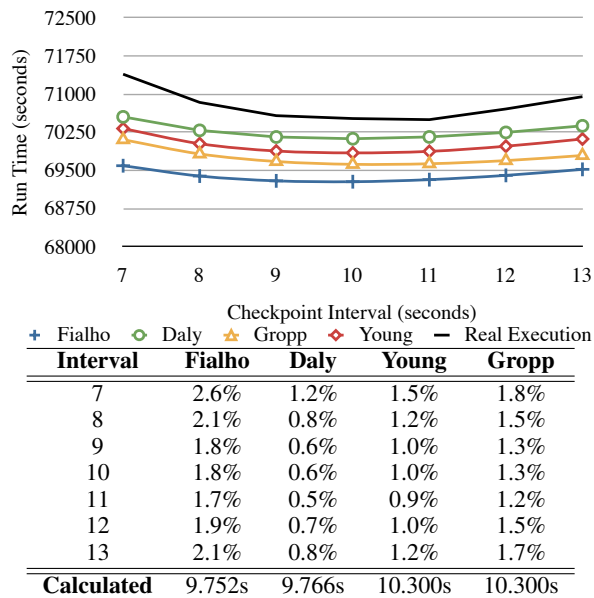


Fig. 3: Comparison of real execution and overhead prediction of the models for $\alpha = 100$, $t_c = 0.530$, $t_l = 0.505$, $t_d = 0$, values in average. Application runs in 62,830 seconds without fault tolerance and in absence of faults. The table shows the relative error of predicted overhead of the models for each checkpoint interval used. The last line presents the optimum checkpoint interval estimation of the models.

of current models is inappropriate in this scenario, the values achieved with other models will be included on the following experiments¹.

To run experiments presented in this section the MTTI (α) has been set to 100 seconds. The RADIC/OMPI library has been configured to send a heartbeat every 1 second. Thus, the fault detection latency (t_d) is 0.5 seconds. As this library performs receiver-base message logging during the recovery phase messages are already available in the log. Thus, the time needed to process the message log (Δ_{lr}) tends to be unappreciable because there is no message replaying [14].

4.2.1 Inter-Process Dependency Factor Effectiveness

To assure that the inter-process dependency factor is the only variable which changes between executions a synthetic application has been designed. The message logging interference and the time needed to take and load a checkpoint are quite similar for the same number of process *per* node.

¹Other models had been designed to be used with serial applications. To compare these models with ours may be unfair while running applications protected by uncoordinated checkpoint combined with message logging. However, it is important to show de benefits of using models specifically designed for uncoordinated checkpointing.

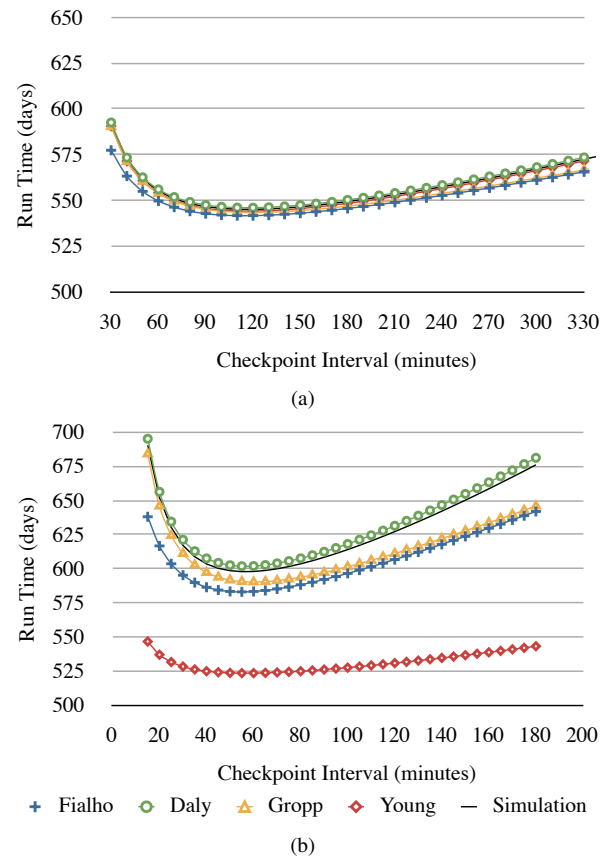


Fig. 4: Comparison of simulation results for values depicted in table 1 using a 24 hours (a) and 6 hours (b) MTTI.

Table 2: Relevant characteristics of the synthetic application used to verify the effectiveness of the inter-process dependency factor. For each model, the first column shows the optimum checkpoint interval calculated and the second column shows the predicted overhead error. $\alpha = 100$ and $t_d = 0.5$. Values are expressed in seconds.

# of Processes	ϕ	t_c	t_l	Δ_{lp}	Δ_{lr}	Fialho		Daly		Gropp		Young	
						σ	Error	σ	Error	σ	Error	σ	Error
4	1.0000	1.630	1.643	2.771	0.000	16.30	3.67%	16.42	2.8%	18.05	4.7%	18.05	3.4%
9	0.5556	1.622	1.596	2.763	0.000	22.39	3.15%	16.39	1.3%	18.01	0.6%	18.01	0.0%
16	0.3125	1.691	1.610	2.765	0.000	31.00	2.21%	16.70	5.3%	18.39	3.3%	18.39	3.3%
25	0.2000	1.650	1.634	2.779	0.000	38.70	2.11%	16.52	7.0%	18.17	5.1%	18.17	4.8%
16 (4×4)	0.3125	4.954	5.131	4.188	0.000	50.46	3.08%	26.52	12.1%	31.48	7.0%	31.48	6.4%
36 (4×9)	0.1389	5.032	5.199	4.160	0.000	78.73	2.65%	26.69	17.4%	31.72	12.2%	31.72	10.4%
64 (4×16)	0.0781	4.981	5.287	4.399	0.000	106.06	1.88%	26.58	20.4%	31.56	15.3%	31.56	12.7%
100 (4×25)	0.0500	5.284	5.330	4.328	0.000	137.76	1.63%	27.22	22.9%	32.51	17.6%	32.51	14.6%

The synthetic application has been programmed using the SPMD paradigm. A computing and a communication phase compose each process. The computing phase is represented by a 2000×2000 matrix multiplication and during the communication phase processes communicate to the right and lower neighbours. These phases are repeated until a defined amount of work has been done. The computing and communication load are the same for all executions. Thus, the interference caused by the message logging is the same in all experiments regardless of the number of processes used. However, the value of the inter-process dependency factor changes accordingly to the number of processes.

Besides other variables, table 2 depicts the value of ϕ for all executions. Values of message logging operation (Δ_{lp} and Δ_{lr}), checkpoint taking (t_c), and checkpoint loading (t_l) are averages of all measurements done during application execution. The value of the checkpoint interval has been previously calculated based on the applications characteristics and is used to configure the RADIC/OMPI library.

As shown in figure 5, as the number of processes increases (or the value of ϕ decreases) so does the accuracy of our model. On the execution with 4 processes the value of ϕ is 1. In this case all models perform similarly. However, as the number of processes increases other models depicts a loss of accuracy. Analysing figure 5(b) it is easy to conclude that previous models cannot be used with parallel applications protected by uncoordinated checkpoints combined with message logging. Especially with a high number of processes.

4.2.2 Correctness of the Message Logging Modelling

These experiments use the LU application from the NAS Parallel Benchmarks [15] running with 8 processes, one *per* node. LU has been executed using class B and C. Table 3 depicts relevant characteristics of LU. Δ values reflect the average measurements done during application execution. The number of iterations of LU class B and C has been modified to 300,000 and 37,500 respectively. The LU application presents a ϕ value equal to 0.5625 for 8 processes.

As shown in figure 6 our model performs better than any

other. Because other models do not consider the message logging time they present an overhead prediction relative error greater than 20% for LU class B. It means that these models are not useful to predict the overhead for parallel applications using uncoordinated checkpointing combined with pessimist receiver-based message logging. Nevertheless, our model presents a modest overhead prediction error for both class B and C of the NAS LU. Notice that from figure

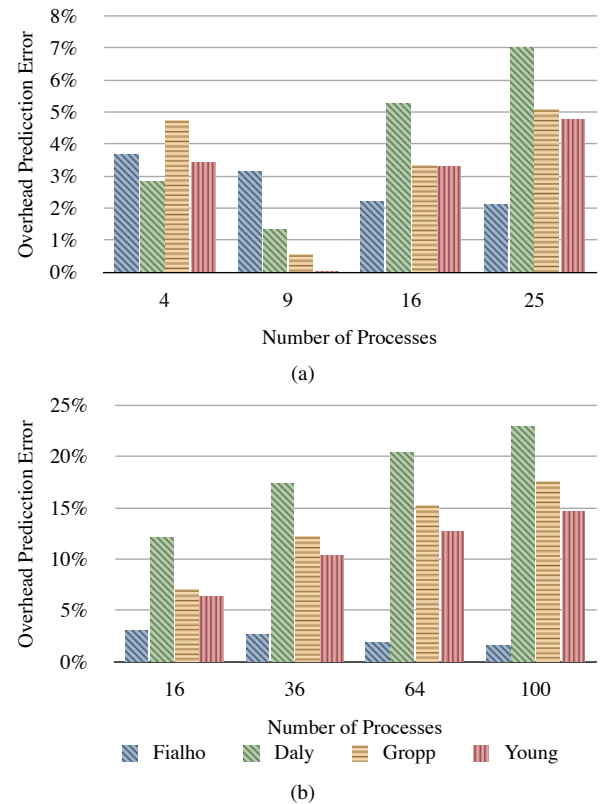


Fig. 5: Overhead prediction error for a synthetic application running with (a) 4, 9, 16, and 25 processes, 1 *per* node, and (b) 16, 36, 64, and 100 processes, 4 *per* node. Values of variables are depicted in table 2, $\alpha = 100$, and $t_d = 0.5$.

Table 3: Characteristics of the NAS LU class B and C. For each model, the first column shows the optimum checkpoint interval and the second the predicted overhead error. $\alpha = 100$, $t_d = 0.5$, $\phi = 0.5625$. Values are expressed in seconds.

LU Class	t_c	t_l	Δ_{lp}	Δ_{lr}	Fialho		Daly		Gropp		Young	
					σ	Error	σ	Error	σ	Error	σ	Error
B	0.605	0.559	38.257	0.005	10.353	3.0%	10.395	25.1%	11.000	25.9%	11.000	22.0%
C	2.057	2.102	13.961	0.007	18.065	0.5%	18.065	5.6%	20.283	8.0%	20.283	5.8%

6(a) to figure 6(b) other models presented a decrease in the overhead prediction error while the opposite occurs with our model. This occurs because the ratio between compute and communication changes reducing the interference of message logging.

5. Conclusions

This paper has presented a novel model to calculate the checkpoint interval to minimise the overhead introduced by fault tolerance on parallel applications.

We have shown that our serial model presents a difference of less than 1.2% from other models on average and the execution time prediction error is smaller than 3% in comparison with a real application execution. With regard to our parallel model, it presents an overhead prediction error smaller than 5% for the applications tested. Furthermore, we have demonstrated that our models perform better when the number of processes increases and there is less dependency between processes.

5.1 Model Utilisation and Future Work

To reduce the number of variables in the checkpoint interval model that minimises the fault tolerance overhead, an analysis of variable sensitivity can be conducted. However, we consider our model short enough to be incorporated into any fault tolerant MPI library. This permits the dynamic definition of the checkpoint interval based on measurements of the time needed to perform fault tolerance procedures. Using this approach, users can achieve better results because the checkpoint interval value reflects the application characteristics at a given moment.

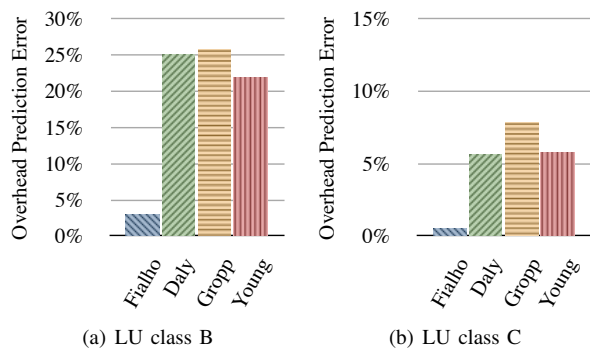


Fig. 6: Model overhead prediction relative error for LU class B and C. Values of variables are depicted in table 3.

Another important issue is to prove that the parallel model is suitable for libraries that implement uncoordinated checkpointing combined with sender-based message logging.

References

- [1] W. Gropp and E. Lusk, "Fault Tolerance in Message Passing Interface Programs," *International Journal of High Performance Computing Applications*, vol. 18, no. 3, pp. 363–372, 2004.
- [2] J. Daly, "A higher order estimate of the optimum checkpoint interval for restart dumps," *Future Generation Computer Systems*, vol. 22, no. 3, pp. 303–312, 2006.
- [3] W. Jones, J. Daly, and N. DeBardeleben, "Impact of Sub-optimal Checkpoint Intervals on Application Efficiency in Computational Clusters," *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pp. 276–279, 2010.
- [4] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes," *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pp. 1–18, 2002.
- [5] A. Duarte, D. Rexachs, and E. Luque, "Increasing the cluster availability using RADIC," *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, 2006.
- [6] J. Young, "A First Order Approximation to the Optimum Checkpoint Interval," *Communications of the ACM*, vol. 17, no. 9, pp. 530–531, 1974.
- [7] Y. Lin, B. Preiss, W. M. Loucks, and E. D. Lazawska, "Selecting the Checkpoint Interval in Time Warp Simulation," *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pp. 3–10, 1993.
- [8] E. Gelenbe, D. Finkel, and S. Tripathi, "Availability of a distributed computer system with failures," *Acta Informatica*, vol. 23, no. 6, pp. 643–655, 1986.
- [9] K. Wong and M. Franklin, "Distributed Computing Systems and Checkpointing," *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pp. 224–233, 1993.
- [10] A. Bouteiller, P. Lemarinier, K. Krawezik, and F. Capello, "Coordinated checkpoint versus message log for fault tolerant MPI," *Proceedings of the 2003 IEEE International Conference on Cluster Computing*, pp. 242–250, 2003.
- [11] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.
- [12] M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.
- [13] L. Fialho, G. Santos, A. Duarte, D. Rexachs, and E. Luque, "Challenges and Issues of the Integration of RADIC into Open MPI," *Proceedings of the 16th European PVM/MPI Users' Group Meeting*, pp. 73–83, 2009.
- [14] S. Rao, L. Alvisi, and H. Vin, "The Cost of Recovery in Message Logging Protocols," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 160–173, 2000.
- [15] W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow, "New Implementations and Results for the NAS Parallel Benchmarks 2," *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

Byzantine-Tolerant Grouping Fault Detection Protocol under High Churn Networks

Huawei Lu¹, B. Shuyu Chen², Xiaoqin Zhang¹ and Guanghui Chang¹

¹ College of Computer Science, Chongqing University, Chongqing, China

² School of Software Engineering, Chongqing University, Chongqing, China

Abstract - *To implement fault detection under large-scale, high churn, strong reliability required network environments, the paper presents a Byzantine-Tolerant Grouping Fault Detection Protocol (Bt-GFDP), which overcomes the problems like network congestion and latency instability in traditional message dissemination systems. Meanwhile it generates self-stabilizing groups which could withstand high churn without affecting the ability of other members to disseminate their messages. The members are robust against Byzantine faults by making use of signatures and random numbers as era. Bt-GFDP was proven to be correct and effective by experiments.*

Keywords: High churn networks; Fault detection; Byzantine-tolerant; Grouping; Detection mode

1 Introduction

The traditional studies on network infrastructure mainly focused on the subjects of the efficiency of information transmission, the integrity of functions and the scalability and extendibility of network structure, while less study interests were put on the dependability of the network. Nowadays a key task of network computing services is to provide trustworthy service results to end users under the threats of intrusions, attacks, and failures in modern network environments. In [1] Professor David Patterson has pointed out, the construction of today's computer system is to provide high-reliable network services. However, the faults on networks themselves and the nodes running on them make networks not dependable.

Highly reliable application systems which support fault detection based on fast developing overlay networks such as Grid, P2P, and Wireless Sensor Networks systems have been brought out^[2-4]. 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, high churn and transmission-uncertainty [5-7]. 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 [8], but it does not consider the problem of heart-beat message transmission, which could lead to the effect degradation when network overhead grows. Fault

interval Δt prediction using linear regression probability was described in [9-11]. 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 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 [12]. This protocol takes the advantage of the high reliability of message dissemination in the network while avoids the problem of network congestion. But too many redundant messages would be generated in the system, which would reduce the extendibility of the system. To reduce the system overhead, a parasitic fault detection algorithm was presented in [13]. It does not produce additional detection message and effectively reduces the overhead of the system. But this introduces tightly coupling between the detection module and the application system, which means that this is not a universal method.

Three trends make Byzantine Tolerant fault detection increasingly attractive for practical deployment. First, the growing value of data and the falling costs of hardware make it advantageous for service providers to trade increasingly inexpensive hardware for the peace of mind potentially provided by BFT replications. Second, mounting evidence of non-fail-stop behavior in real systems suggest that BFT may yield significant benefits even without resorting to n-version programming. Third, improvements to the state of the art in BFT replication techniques make BFT replication increasingly practical by narrowing the gap between BFT replication costs and costs already being paid for non-BFT replication.

A significant challenge of overcoming churn and facing Byzantine faults is to find ways to limit the ability of faulty members to take advantage of high churn to destroy the system's structure. A random dissemination fault detection protocol which tolerates nodes' Byzantine faults based on flexible grouping is presented in this paper. This work is based on the previous work GFDP^[13]. It takes the advantage of the reliability of Gossip-style dissemination, and reduces the network overhead and time consumption as every group is comparatively autonomous. This protocol possesses the characteristics of high extendibility, low consumption, nodes' Byzantine-tolerance and etc. In section 2 the system model is introduced, and the protocol is discussed in section 3, and experimental results are presented in section 4, and a discussion is given at the end.

2 Definitions

2.1 System model

A network system consists of a limited set of multiple members:

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

m_i in Π here is an abstract presentation of a module or a process, or even a node in the dependable network system. And members may be active or passive.

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

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

Elements m_i and d_j in up two sets:

$$\forall m_i \in \Pi, \exists d_j \in \Omega, \text{ for } i = j$$

d_j is called a fault detector attached to m_i . An active member means its detector participates in the protocol; while a passive one means its detector may be crashed or detached. In the following description, we use “member” and its attached “detector” as the same meaning.

Some of the active members may be Byzantine. We assume that the members may dynamically fail; failed members may recover and need to be re-integrated in to the system.

We assume the existence of a public key cryptography scheme that allows each detector to verify the signature of each other detector. We further assume that non-Byzantine members never reveal their private keys, such that faulty detectors can't forge signatures.

A detector is attached to a specified member in the network system, and all the detectors form a detection set, for $\forall m_i \in \Pi, \exists d_j \in \Omega, i \in N$. Every detector has an identifier $d_j \cdot id$, which is assigned by a central authority (CA). As high churn (members coming and going) in the system and uncertainty about the states of other detectors, each detector maintains temporary lists, as described in section 3.1.

A correct member has an internal timer that runs at a bounded drift from real time, which enables its detector to measure periods of time with relative precision. A global clock is not required to be synchronized.

Definition 1. An active member is correct if its attached detector is active and following its protocol, processes messages in no more than π real-time units and has a bounded drift of the member's internal timer. An active member that is not non-faulty is considered Byzantine.

A member or its detector will be called faulty or Byzantine, interchangeably.

A member that faces transient failure may find itself in an arbitrary state. Therefore it may take some time to integrate itself into the system.

Definition 2. A communication network is non-faulty if messages arrive at their destination within δ real-time, and

the content of the messages as well as the identity of the sender are not tampered with.

Once the detection system is coherent, a message between any two correct members is sent, received, and processed within Δ real-time units, where Δ includes π , δ and drifts of local times.

2.2 Byzantine fault types

In general, a truly decentralized Byzantine fault tolerant (BFT) system should meet the requirements of the cost of BFT replication for $f = 1$ failures with 4 agreement nodes and 3 execution nodes [3]. But in our work, we fully take the advantage of using CA, which could reduce the difficulty of identifying Byzantine members. We defined two kinds of Byzantine faults in our system.

A message-denying fault issued by an individual member is defined as a Black Hole fault. In addition, two or more faulty members may collude together to form a larger “Black Hole”. These two faults are called as Black Hole Class (BHC) faults. The BHC fault means the members become dumb as either their detectors crashed or being attacked by malicious users. BHC faults become more serious if faulty members participate in message propagation at early gossip rounds, or a large number of faulty members collude together. Under such conditions, BHC faults largely reduce the message number, and thus seriously slow or even cease message propagation.

Moreover, a Message-Faking (MF) fault is more harmful than a BHC fault. Unlike BHC faulty members that only deny messages, an MF fault directly propagates incorrect information, misleading other members to make a wrong decision. As we make use of digital signatures, the tampered messages would be recognized by correct members. But since members may fail and recover, Byzantine members can replay old signed messages, as the MF faults in our work. In a self-stabilizing environment it is challenging to identify replayed messages.

3 Protocol specification

3.1 The member views

Every detector in the network system would disseminate its *Heart Beat* message to some members in its healthy members list or suspicious members list initiatively and randomly in every time interval Δ . A member's *Beat Counter* would add one during every dissemination round. After received this message, every other detector would record received time stamp *Last time*, set suspicious time interval 2Δ , and wait for the next coming message. If no new message came from the member being detected in time *Last time* + 2Δ , or the message itself was not *Beat Counter* + 1, or the message was not certificated, it indicates that the detected member may be faulty.

Bt-GFDP detectors gossip in the group on the dissemination structure, which is discussed below, and the connectivity of the group is determined so with high probability all detectors learn of new gossip within Δ time units.

Every detector would maintain a list $d_i \cdot Heal$, which generally includes all correct members in its group. And if a group member is suspected as failed or faulty, it will be add into the list of recently suspected members $d_i \cdot Susp$, and if it does not recover in 2Δ time units, it would be add into the list of recently removed members $d_i \cdot Remd$, or it would be back into $d_i \cdot Heal$. The member in $d_i \cdot Remd$ is still considered as potentially connected if it rejoins the system correctly in Δ and it would be moved into $d_i \cdot Susp$, or it would be removed permanently. Any new joiner will be in the list of recently accepted members $d_i \cdot Join$ for Δ time units, giving the rest of the active group members a chance to identify the new addition, and then will be integrated into $d_i \cdot Heal$. And all above 4 lists constitute the view of d_i .

All the lists could be customized by a structure. The list keeps a *Beat Counter* and group member IDs. An ID is a unique member identification with assigned by CA, generally includes address information, and *Beat Counter* has the meaning as a normal detector counter.

Lemma 1. A crashed or BHC member will be removed from the view of every correct and active member within 3Δ time units.

It does not reach agreement on views in Bt-GFDP, therefore the views can always differ. At steady state the difference among the views

3.2 Group initialization

With the growth of the network size, the amount of fault detection messages would grow drastically and the latency would be unbearable, and the inaccuracy of the system would be accumulated, which would lead to detection method fails partially. To solve this, a grouping mechanism is used in this paper, by which all members are divided into flexible autonomy groups. And the growth of the network size would not effect upon problems.

If one node I cannot be integrated into any group, it would trigger a group initialization process as an initial member. For different practical achievement, the number of groups denoted as α can be determined by the time consumption Δ based on the actual network environment and accuracy requirement. And the optimal number of members in a single group is denoted as β , and we have $\beta = \lceil N/\alpha \rceil$, here N is the total number of ungrouped system members. And in some high churn situations, the group size would be smaller than $\lceil \beta/2 \rceil$, then the group would be dismissed, and the remaining active members in that group would be integrated into other groups whose members are less than the maximum size 2β , or even the group initialization process would be triggered to form another detection system, this is the group separation process, which would be discussed in section 3.3.

The procedure of the group construction is as follow.

Algorithm 1:

Step 1: As the initial member, I firstly request the CA for the N ungrouped members and send out $N - (r - 1) \times \beta - 1$ broadcast detection messages to ungrouped members. Here r is the broadcast round. And we can get clearly that the number of broadcast messages is $N - 1$ at round 1.

Step 2: Every healthy ungrouped member responds to this broadcast message, and initial member I sort the responses based on time, choose $\beta - 1$ members as a group whose center is the initial member I . Then I is chosen as an agent member.

Step 3: Choose the member whose response is the latest from the remaining members which exclude the $\beta - 1$ members in the former step as next initial member. Add 1 to r then return to step 1 and 2.

Step 4: When total number of remaining nodes is no more than β , then the last agent member would multicast to them that they could join group freely.

In this algorithm every time the initial node is chosen, then a new group's agent member is decided. Whenever there's any living message passing, it will be through agent members between the groups.

Lemma 2. When any healthy member in the system executes *Algorithm 1*, the system would be divided into several groups which cover all healthy members in the system in finite steps, and the group size would be less than 2β .

3.3 Group separation

The group size cannot be aggregated without limitations. When the group size grows up to 2β , the process of group separation would be trigger by the agent member.

The process of group separation is described as follow.

Algorithm 2:

Step 1: As a group whose size is up to the threshold of 2β , its agent member I firstly broadcasts a query message to all of its neighbors. Here we get clearly that the number of broadcast messages is $2\beta - 1$,

Step 2: Every healthy in-grouped member responds to this broadcast message, and the original agent member I sort the responses based on time, choose the first $\beta - 1$ members as a group whose center is the original agent member I .

Step 3: Choose the member whose response is the latest from the remaining healthy members which exclude the $\beta - 1$ members in the last step as next agent member. Then gossip the separation information to them. The separation information includes the changed agent member and a list of remaining ungrouped members.

Step 4: When a member gets the separation information, it would disseminate the message to its old neighbors and then compare the members list in the separation information with it local view. If it is in the list, which means it's not grouped in

Step 3, it would change its agent member and keep the list as its new. If it is not in the list, it would keep its agent member and generate a new view by deleting all the members in that list from its local view.

3.4 The era list

The assumed existence of digital signatures reduces the ability of Byzantine members to mislead correct members. But since members may fail and recover, Byzantine members can replay old signed messages. In a self-stabilizing environment it is challenging to identify replayed messages.

In order to reduce the ability of Byzantine members to perform a convincing replay attack, a member needs a mechanism that produces some randomization to its new identity when it recovers. To achieve that, a member chooses periodically a new symbolic number. A new era is the signed pair (pre_sym , new_sym), where pre_sym is its previous symbolic number and new_sym is its new symbolic number. The values of symbolic numbers are random numbers from a large enough space, which is much larger than the memory space of the faulty members, so that the probability that a member repeats the pair (pre_sym , new_sym) is theoretical negligible, and the ability of a faulty member to replay such a pair is even smaller. And such error could be ignored in the system.

The introduction of a random era is similar to choosing a random id. Therefore, in our protocols, whenever a member sends out a signed message it should include its current era. We assume that members disseminate signed messages, and members ignore any message that is not signed properly or does not carry the matching era.

Each member maintains as part of its view the latest era of each member in the view which described in Section 3.1. A receiver of a signed message will consider the message current only if the era matches the last era the receiver knows of. If the recent era was received less than Δ ago, it can still accept signed messages containing the previous era value. The message is current also when the member did not receive the new era yet, but its latest copy of the era matches the pre_sym part of the era of the received message. When a member d_i updates its era, it will send a special message containing the new era; this message is disseminated the same as other messages in the system. If a member d_j receives such a message and d_j 's current era is equal to d_i 's pre_sym then d_j updates its view of d_i 's era.

4 Experiments

The experimental environment is composed of three resource sites in China Education and Research Network (CERNET), including 20 nodes at National Linux Technology Center (LinuxCenter) and 10 nodes at Network and Distributed Computing Lab (NetmobiLab) in the School of Software at Chongqing University (CQU), and 8 PCs (running Linux 2.6 Kernel) at Modern Service Computing Lab in the School of Computer Science University of Electronic Science

and Technology of China (UESTC). Every node at LinuxCenter and NetmobiLab is configured with P4 2.4G CPU and 512 MB RAM, and the OS running on is Ubuntu 9 with 2.6.20 kernel. Each node at these three sites is interconnected by 100 Mb Ethernet.

The module was implemented in our development toolkit and tested in WAN to simulate the random dissemination fault detection. UDP is used by message dissemination inside the group and TCP is used between the groups.

First we set a node in NetmobiLab as CA server, and the CA server never malfunction in our experiments. Then based on the differences of hardware platforms, we were running 2-16 independent processes on each node to simulate an individual entity in the network. And finally an overlay network with a configurable entities number from 2 to 500 was constructed.

4.1 Experiments on coverage of Bt-GFDP

To testify the availability of Bt-GFDP under different group size, we created three networks with different sizes. And we configured all nodes in one network to be in just one group. The fan-out number was set to 1 and the infected nodes disseminated messages in every round. The coverage of infected group members under different configurations was illustrated in Fig 1. And the group size was denoted in the figure as 64, 128 and 256 respectively.

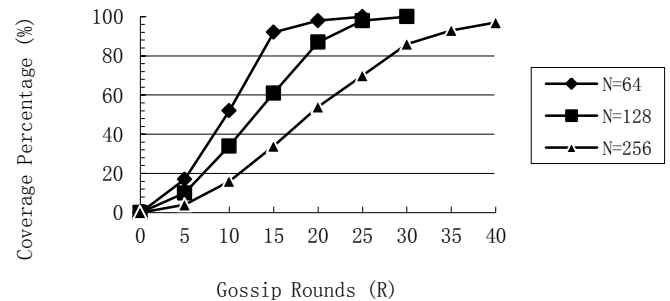


Fig 1 Bt-GFDP coverage under different group sizes

From the experimental data, we could see that the coverage of Bt-GFDP was eventually complete when the Gossip rounds were large enough, and the coverage ratio increased dramatically in the midway, but time consumption increased greatly with the expansion of the system size, as the gossip process is probabilistic. The result thus reveals that the larger the network size is, the more time consumption is. And the correct configuration of group size and fan-out number is crucial to the coverage ratio and dissemination rounds.

4.2 Experiments on availability of Bt-GFDP

We simulated the Bt-GFDP in a 100-node system, and all these nodes are in the same group and every node is reachable to others. In this stage we screened the function of group separation.

We considered a scenario where BHC attacks resulted in message loss during propagation. Figure 2 shows the nodes' stable states during the process of group initialization under

different scenarios. The fan-out number was 2 and the number of gossip rounds was 30.

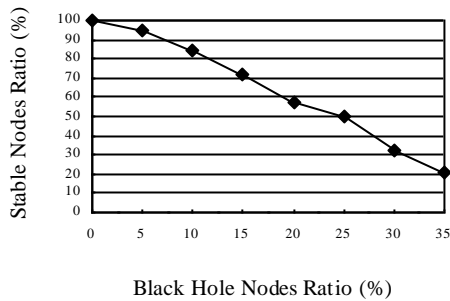


Fig 2 The curve of stable nodes ratio

Fig 2 shows the Stable Nodes Ratio (SNR, which demonstrate the ratio of nodes who can distinguish BF nodes from other nodes, means the percentage of nodes that are in a stale group) along with the Black Hole Nodes Ratio (BHNr, which also can be directly considered as BF nodes ratio in the network). We observe that almost all the BF nodes could be recognized by healthy nodes and the healthy nodes could be in a stable group when BHNr is less than 10%. And it's logical that SNR decrease when BHNr increases. But when BHNr reaches 35%, only about 20% of nodes could remain stable, this means that other 45% nodes are not in a stable group though they are healthy. As too many faulty nodes would break the balance of group maintenance, groups would easily be crashed and triggered separation process. And it got almost

Tab 1 Comparison of 3 Algorithms

D \ P	Day1	Day2	Day3	Nig1	Nig2	Nig3	\bar{C}	e
Bt-GFDP	1032	1120	1074	996	984	964	1208	0.04
GFDP	820	856	923	792	813	756	826	0.17
Renesse	1228	1356	1303	1084	1101	1205	1212	0.33
Flood	182	168	171	183	179	182	177	0.48

In the Tab 1, D means different experiment dates, \bar{C} is average overhead, e is the false alarm rate. We can see from the table that the network overhead of Bt-GFDP is 30% higher than GFDP's and keeps almost the same as Renesse's, but the false alarm rate is dramatically lower than the others. Comparing with GFDP, the reason for the higher network overhead is the group information between the members, and lower false alarm rate is due to its mechanism of Byzantine tolerance, and that's important to time prediction in fault detection. Therefore, the false alarm rate is comparably low in the daytime. Though the Flood method takes advantage of message amount, it is not as effective as those two methods above, especially at some busy hours, the network congestion and package dropping brought out by this method, it turns out to be not available. And the reasons why our Bt-GFDP did comparably well on network traffic controlling are not only the redundancy avoidance policy but also the effect of group division.

the same result during the Bt-GFDP running process other than initialization process.

This demonstrates that Bt-GFDP function well when BHNr is less than 15%, only less than 10% healthy nodes are in unstable state. Moreover, the performance drops dramatically when BHNr grows larger. So this is another improvement we should consider in our future work.

4.3 Comparison with other works

In this part we configured a network with 192 members, and for Bt-GFDP we set $\beta = 64$. So for Bt-GFDP we could get 3 group topologies. A message counter C_i was set in every member, whenever the member received a message, the counter is triggered. When the system reached the coverage threshold, we could calculate the total system overhead:

$$C = \sum_{i=1}^{192} C_i$$

Moreover, we set the BHNr to 15%, and the faulty members were randomly generated and they could recover with certain probability. As actual network would work differently at different time period of a day, we did 3 separated experiments under same conditions at different times.

In actual calculation, we firstly calculated the overhead in a single group and then the total overhead. The comparison of the experiments is as follows:

5 Conclusions

To meet the requirements of large-scale, high-churn, nodes' Byzantine faults, and uncertainty latency of message dissemination, Bt-GFDP is proposed in this paper. The simulation experiments show that: Bt-GFDP could effectively control the redundant network messages, and it is better at time consumption, and it can offer more precise false alarm. In our future research, we would like to construct more configurable detection group and compatible malicious members which fakes messages. And the role of CA should be weakened and the detection should be accomplished by the cooperation of healthy members. Moreover, the mechanism of fan-out number and group size configuration should be further studied in the future.

6 References

- [1] D. Patterson. "Recovery oriented computing", Presented at Princeton University. <http://roc.cs.berkeley.edu/talks/UIUC.ppt>, 2002

- [2] T. D. Chandra, S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2) : 225-267, March 1996
- [3] J. Yin, J. P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proc. SOSP*, October 2003
- [4] 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
- [5] 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.
- [6] 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
- [7] 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
- [8] 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.
- [9] S Hwang, C Kesselmanl . "A flexible framework for fault tolerance in the grid", *Journal of Grid Computing*, Vol.1, Issue No.3, 251-272, 2003
- [10] Dong Tian, Shuyu Chen, Feng Chen. "A Dynamic Fault Detection Algorithm under Grid Environments", *Journal of Computer Research and Development*, Vol.43, Issue NO.11, 1870-1875, 2006 (in Chinese).
- [11] T D Chandra, S Toueg. "Unreliable failure detectors for reliable distributed systems", *Journal of ACM*, Vol.43, Issue2, 225-267, 1996
- [12] W Chen, S Toueg, M K Aguilera1. "On the quality of service of failure detectors", *IEEE Trans on Computers*, Vol.51, Issue No.2, 13-32. 2002.
- [13] Guanghui Chang, Huawei Lu, Shuyu Chen, Ishiang Shih. "Grouping Fault Detection Protocol under Dynamic Network Environments", *PDPTA (2010)*
- [14] N Hayashibara, X D'efago, R Yared, et al. "The ϕ accrual failure detector", *Proc of the 23rd IEEE Int'l Symp on Reliable Distributed Systems (SRDS'04)*. Los Alamitos, CA:IEEE Computer Society Press. 66-78, 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.
- [16] Chaoshu Zuo, Xinsong Liu, Yuanjie Qiu, et al. "A Node Fault Detection Algorithm in Distributed Parallel Server", *Journal of University of Electronic Science and Technology of China*, Vol.36, Issue No.1, 119-122, 2007 (in Chinese).

SESSION

PARALLEL COMPUTING IN CLUSTERS: OPTIMIZATION AND PARALLELIZATION OF SEQUENTIAL APPLICATIONS

Chair(s)

Prof. Fernando G. Tinetti

Scalability Analysis of a Parallel Dynamic Data Driven Genetic Algorithm for Forest Fire Spread Prediction*

Mónica Denham¹, Ana Cortés², Tomás Margalef²

¹Universidad Nacional de Río Negro, Sede Andina, Ingeniería Electrónica.
San Carlos de Bariloche, Río Negro, Argentina.

²Departament d'Arquitectura de Computadors i Sistemes Operatius, Universitat Autònoma de Barcelona,
08193 - Bellaterra (Barcelona) Spain

Abstract—*This work presents a performance study of a Parallel Dynamic Data Driven Genetic Algorithm (Parallel DDDGA) for Forest Fire Prediction. The main objective is to obtain a trade off between prediction quality and the time incurred in that prediction. For this purpose, High Performance Computing is applied to exploit the parallel features of the proposed parallel DDDGA. A framework was developed, including a data driven method where suitable data is injected and Genetic Algorithm convergence is accelerated. Through this work, application time reduction and application scalability is studied.*

Keywords: Parallel programming, Efficiency evaluation, High Performance Computing, Workload balance, Forest Fire Prediction

1. Introduction

Several times, forest fire predictions are not precise as we would like, due to many uncertainly sources: fire behavior models, landscape representation, hardware and software restrictions, imprecise input parameter values, etc, [1] [2] [3] [4].

Not only wildland fire environment (topography, weather conditions, fuel features) influences fire spread, but also the fire itself resulting in a very complex fire behavior pattern.

Nowadays, simulation techniques are used to predict forest fire behavior. A classical prediction schemes uses a given forest fire behavior simulator feeding it with the “available” input parameter values generating a forest fire spread evolution.

The main problem when applying this classical prediction method is the low quality of the prediction results. Some reasons of such a misleading results are the inaccurate use of the underlying simulators, a poorly interpretation of the results, incomplete or inexpert simulator operation, etc. Furthermore, applications for forest fire simulation are themselves no exempt of inaccuracy (they are implementing a complex phenomenon, resulting in complex mathematical and physical formulation, and finally, a complex computer code). In addition, geographical information complexity,

which is modeled by maps, those which are considered as “true information”, can also include some limitations [3].

Simulator input data is another uncertainty source ([1]). There are certain parameters that cannot be measured directly, so, they are estimated from indirect measurement techniques. Other parameters can be measured in certain particular points but the value of such parameters must then be interpolated to the whole terrain. Additionally, maps used for describing topography, fuel, moistures, etc. are divided in cells and values into a cell are constant (usually they are the predominant value for the characteristic, in other cases, it is the averaged value). Furthermore, this maps are updated in certain time recurrence, so it is easy to use neither space nor time actual values.

Another important error source is the nature of input parameter values. Some parameter values are constant through fire life, such as terrain features, but another parameter values are dynamic and their values change through time (wind direction and velocity, fuel moisture, etc.). So it is very difficult to have the actual value for these parameters through space and time [4].

Identifying main forest fire prediction uncertainly sources, a new method is proposed, where input parameter values are pre-searched and improved, obtaining more accuracy simulations. A *Parallel Data Driven Genetic Algorithm* is proposed and developed.

Furthermore, response time is crucial through current work field: a fire progress prediction is really useful when it is available in a very short period of time. Thus, response time requirements forces us to obtain a high performance method to predict forest fire behavior.

Predictions quality improvement when proposed method is applied is study through [5] and [6]. In this work, parallel method main characteristics are introduced, and then, a method scalability study is reported.

2. Parallel Data Driven Genetic Algorithm

The proposed framework consists on two stages: Calibration Stage and Prediction Stage (depicted in figure 1). Three instants of time are required for obtaining the final predic-

*This research has been supported by the MICINN-Spain under contract TIN2007-64974

tion: Calibration Stage performs simulations from instant t_i to t_{i+1} and then, prediction is executed from instant t_{i+1} to t_{i+2} .

During Calibration Stage, all possible combination of forest fire simulator inputs are considered (simulator *fireLib* was used as simulation engine [7] [8] [9]). Input parameter combinations are selected by a Genetic Algorithm [1] [10], which selects the best combinations for simulator input parameters through its iterative generations. Then, a feedback process is executed, where simulations are compared against real fire progress. This comparison allows Genetic Algorithm to evaluate input parameter fitness and improve its characteristics (by genetic operators: selection, elitism, crossover and mutation).

Forest fire simulator *fireLib* uses a cellular automaton model in order to represent fire progress over the land. Real fire line is available and represented by a map of cells (this real fire line holds fire state for time t_{i+1} , necessary for this two stages prediction method). Our Genetic Algorithm fitness function (called *error function*) is based on a cell by cell comparison of simulated map cells and real fire line map cells. Thus, error function determines differences between real fire line and simulated fire line. Genetic Algorithm intends to minimize this error function.

Once Calibration Stage is performed, the best individual (input parameter combination) obtained is used through Prediction Stage, in order to produce the final predicted propagation.

Figure 1 shows the proposed methodology, where *RF* and *SF* mean Real Fire and Simulated Fire respectively.

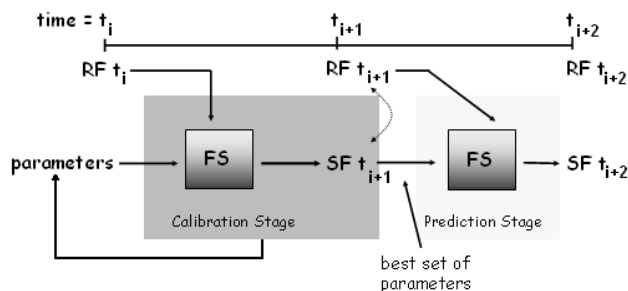


Figure 1: Two stages method for forest fire propagation prediction.

Due to proposed method requirements, real fire progress is available for instant t_{i+1} . Then, this real fire map is analyzed in order to obtain useful fire progress information, which is used for steering Genetic Algorithm execution. Thus, fire line is analyzed and main fire line characteristics are obtained: fire velocity, fire spread main direction and distance covered by fire progress.

Slope inclination and aspect are non dynamic input parameters. Neither of them alter its characteristics through wild fire life. Moreover, current massively geographical

information availability allow us to consider slope and aspect as fixed and known input parameters values through our application.

Then, real fire line direction and velocity are known, as well as slope main characteristics.

In wild fires field it is well known that wind and slope features combination determines fire progress, which is mainly characterized by fire spread direction and maximum spread velocity.

Then, having fire and slope main characteristics our Dynamic Data Driven Algorithm calculates wind features to achieve fire progress similar to studied one. It means, the method finds the specific wind necessary to achieve real fire progress in a known slope and aspect conditions. Then, ideal wind direction and speed are obtained, and these wind values are injected during Genetic Algorithm execution.

Thus, taking into account the calculated wind values, two steering methods were proposed: Computational Method and Analytical Method. Analysis and tests of their efficiency are out of the scope of this paper and are widely studied in [5] [6]. The goodness of these methods in terms of prediction quality compared to the classical prediction scheme has been also demonstrated.

The goal of this work is to expose performance and scalability analysis of the application. Next section will depict parallel method main features. Then, execution time and application scalability will be analyzed too.

3. Parallel Dynamic Data Driven Genetic Algorithm

As it is well known, Genetic Algorithms works in an iterative way over a population of individuals. As we had mentioned, through our application each individual consists of a combination of simulator input parameter values.

The initial population (typically generated in a random way) is evolved: Genetic Algorithm applies selection, elitism, crossover and mutation operators to provide a new population that minimizes/maximizes a fitness function. This process is repeated a certain number of times or until the optimum solutions have been reached.

For the case of forest fire spread prediction, the search space of a Genetic Algorithm consists of all possible combinations of the set of input parameters of the underlying simulator. In our particular case of forest fire spread simulation, the elements of the search space (individuals) are all independents, it means, there does not exist relationships or dependences between them. Then, neither time nor order in which simulations are performed are relevant.

Different application task times were evaluated using the monitoring tool MPE Open Graphics ([11]). This study shows us that fire spread simulations are the processes with most CPU time requirements from the whole prediction

scheme.

Figure 2 is one of the MPE graphic obtained, focused on a specific interest zone: fire progress simulation, simulated map analysis and fitness calculation. During this execution, just one process performs all these tasks. As figure legends depict, smooth gray zones are fire progress simulation, while simulated map analysis and fitness (error function) calculation are vertical dark lines. Map analysis and fitness operation are short enough to appear just as a vertical line through application execution pattern. Then, it is straight forward that fire spread simulation is the task with most execution time requirements. Moreover, during Genetic Algorithm progress several simulations are executed.

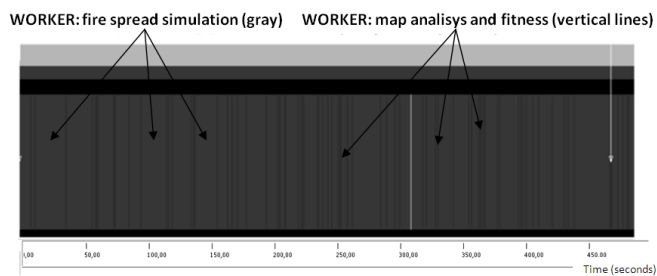


Figure 2: Application time requirements.

Taking into account these characteristics (response time limits, non simulation dependences and simulation time requirements) master/worker programming paradigm was used for parallelized purposes. In the proposed master/worker scheme, the master process performs Genetic Algorithm operations and distributes population individuals among worker processes. Each time a worker process receives an individual, it performs the fire progress simulation and calculates the error function for the individual. Figure 3 illustrates the implemented parallel master/worker application.

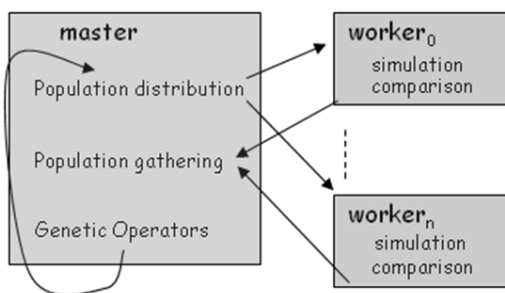


Figure 3: Master and worker processes.

Due to master/worker communication pattern, all individual transmissions have as either source or destination the master process, therefore, the master process communication service can become an application bottleneck.

In order to avoid this potential problem, individuals are distributed by groups (chunks) instead of individual transmissions. When a worker finalizes the evaluation of a specific chunk, this worker process returns the evaluated chunk to the master process. Then, master process sends another non evaluated chunk to it until all chunks are evaluated.

As we had mentioned, due to our application characteristics, workload is performed by the distribution of chunks of individuals. In this way, worker processes perform the fire spread simulation over all cells of the terrain. Moreover, worker processes have real map at instant time t_{i+1} in order to perform error function by the maps comparison.

Next section will show experimental results. Several tests were executed, and the evaluation of different method features was obtained. Next sections will show a group of representative tests and a brief analysis of each one will be given.

4. Experimental Results

The experimentation conducted in this section has been oriented to study two aspects of the parallel DDGA: the convergence of the method and its scalability.

4.1 Convergence Analysis

The convergence analysis of the Genetic Algorithm has been performed by comparing the plain Genetic Algorithm strategy where no steering strategy is applied against the guided Genetic Algorithm where the above mentioned steering strategies (Computational and Analytical) are applied.

This case was performed using a synthetic map, which lasts 14 minutes, divided in 6 time steps of 2 minutes each of them. Map dimensions are $110 \times 110m^2$, divided by $1m^2$ cells. 400 individuals populations were used. Figure 4 shows different method convergences when Genetic Algorithm evolves 10 times the population.

Figure 4 depicts Computational Method convergence. It is straight to see that Computational Method achieves good individual values through its first iterations. More iterations are needed for other Genetic Algorithm configurations in order to achieve high quality simulations. Then, when Genetic Algorithm denotes a faster convergence, less iterations are necessary to achieve good results. In this way, all method response time is reduced, while good quality simulation remains kept.

Several cases had shown the same behavior, so this characteristic can give us the idea of "stability" when Genetic Algorithm is guided toward correct results.

Next figure (figure 5) shows response time analysis. Population sizes and genetic algorithm iterations were fixed while number of processor were varied: 1, 2 and 4 processors were used for current experiment (next subsection will depict more detailed test, where a large cluster was used).

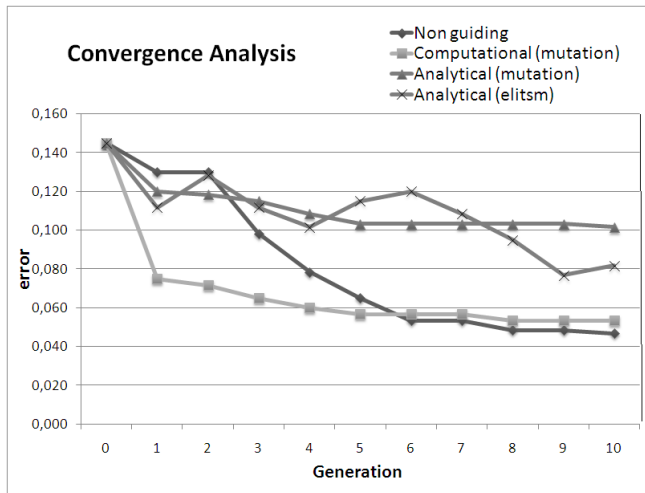


Figure 4: Steering Method convergences through 10 genetic algorithm evolutions.

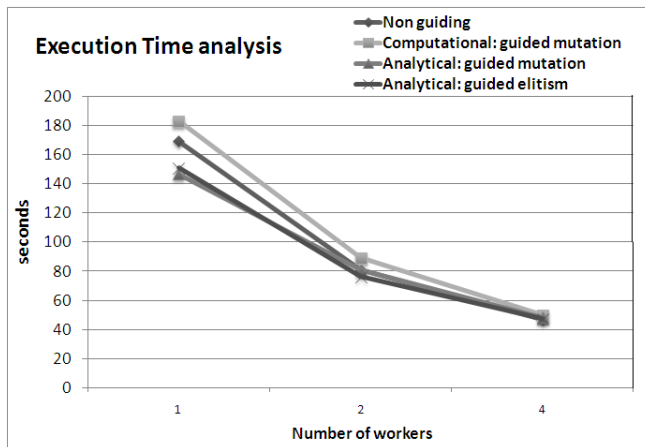


Figure 5: Application scalability for 1, 2 and 4 processors.

As figure 5 shows, execution time decreases when number of computational resources increases. Different proposed methods show similar behavior, then, application scalability is suitable when few processors are used; certainly, this test may correspond to a real use of our application: final user can be a fire department or some governmental department, where computational resources can be few interconnected nodes.

For all tests, used cluster has a queue system that guarantee that when a task is assigned to be executed, this task obtains needed cluster resources in an exclusive form. Thus, a correct time analysis is possible.

4.2 Scalability Study

In this test a real case of a prescribed burning was used. Fire line grows in a wide line, and it lasts 14 minutes. Populations with 512 individuals were used for each test (excepting the last test, where population of 527 individual

were used). Genetic Algorithm was 5 times evolved in each case. In order to increase execution requirements, time step lasts from minute 2 to minute 10, provoking larger simulations.

Execution time is analyzed when 1, 2, 4, 8, 16, and 31 workers were used (32 nodes's cluster). Computational Method times and non guiding times are compared. Number of workers is varied in horizontal axis and execution time is presented through vertical axis of figure 6. Last test uses 31 workers in order to avoid concurrent processes at the same node (that means the master and a worker running at the same node).

Figure 6 shows master and workers execution and communication times. Worker times are the average of the different worker times (except when just 1 worker is used).

Execution times had demonstrated that the application is scalable when the number of processors increases. In addition, Computational and non guiding methods had very similar behavior, then we conclude that proposed steering method does not add extra execution time. Computational Method time requirements are not significant.

Furthermore, communication time is very low for all cases, the use of a big number of worker processes does not increase communication time. Then, application performance was not penalized by communication. In addition, having low communication times implies that load balance is achieved, workers do not spend time waiting for receiving new chunks (group of population individuals).

5. Conclusions

A Parallel Dynamic Data Driven Genetic Algorithm was developed and aspects as communications, load balance, scalability, etc, were carefully studied.

The master/worker programming paradigm has been used to implement the parallel DDDGA for forest fire spread prediction. The application tasks time requirements were studied, concluding that the specific Genetic Algorithm operators like selection, crossover and mutation have not significant time impact in the whole prediction process.

Forest fire simulations have arisen as the most significant task in terms of time requirements. Furthermore, taking into account that the genetic algorithm runs a considerable number of simulations, to reduce the time incurred in executing all these simulations is one of the main objectives to tackle. For this reason, fire progress simulation and evaluation were chosen as the set of tasks to be distributed through all available worker processes.

Due to master/worker communication pattern, genetic individual transmissions could became an application bottleneck. In order to avoid communication bottleneck performance penalties, Genetic Algorithm individuals were divided into chunks and these chunks were dealt through worker processes.

Taking in mind that each simulation has its own time

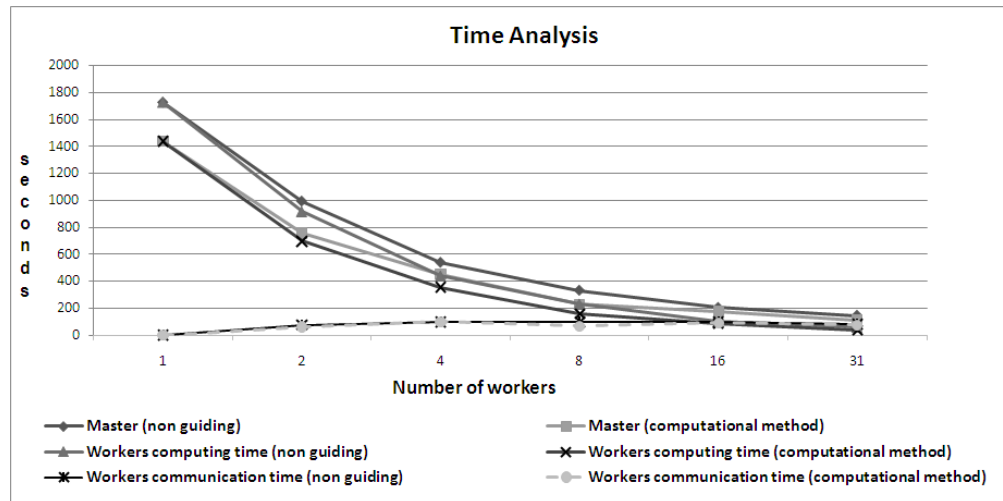


Figure 6: Application scalability for 1, 2, 4, 16 and 31 processors.

requirements (it depends on fire progress, distance and velocity), chunks were distributed on demand, that means: when a worker ends the evaluation of a chunk, master process sends a non evaluated chunk to this worker (until all chunks are evaluated).

The analysis of worker process times and total application response time, allow us conclude that correct load balance was achieved.

On the other hand, Computational and Analytical Methods accelerate Genetic Algorithm convergence. Using real fire progress knowledge, Genetic Algorithm is guided forward good individuals. Then, few Genetic Algorithm iterations are enough to achieve good results. In this way, iterations can be reduced in order to accelerate prediction process (decreasing application response time). Finally, the parallel DDDGA scalability was tested providing satisfactory results.

References

- [1] B. Abdalmaq, A. Cortés, T. Margalef, and E. Luque, "Enhancing wildland fire prediction on cluster systems applying evolutionary optimization techniques," *Future Gener. Comput. Syst.*, vol. 21, no. 1, pp. 61–67, 2005.
- [2] B. Abdalmaq, "A methodology to enhance the prediction of forest fire propagation," Ph.D. dissertation, Universitat Autònoma de Barcelona, June 2004.
- [3] D. Caballero, "Taxicab geometry: some problems and solutions for square grid-based fire spread simulation," in *V International Conference on Forest Fire Research*, D. Viegas, Ed., 2006. [Online]. Available: <http://www.gnomusy.com/Presentations/Taxicabhtm>
- [4] G. Bianchini, "Wildland fire prediction based on statistical analysis of multiple solutions," Ph.D. dissertation, Universitat Autònoma de Barcelona, July 2006.
- [5] M. Denham, A. Cortés, T. Margalef, and E. Luque, "Applying a dynamic data driven genetic algorithm to improve forest fire spread prediction," in *International Conference in Computational Science ICCS 2008*, ser. Lecture Notes in Computer Science, M. Bubak, G. van Albada, J. Dongarra, and P. Sloot, Eds. Springer Berlin Heidelberg, 2008, vol. 5103, pp. 36–45.
- [6] M. Denham, A. Cortés, and T. Margalef, "Computational steering strategy to calibrate input variables in a dynamic data driven genetic algorithm for forest fire spread prediction," in *International Conference in Computational Science ICCS 2009*, ser. Lecture Notes in Computer Science, G. Allen, J. Nabrzyski, E. Seidel, G. van Albada, J. Dongarra, and P. Sloot, Eds. Springer Berlin Heidelberg, 2009, vol. 5545, pp. 479–488.
- [7] C. D. Bevins, *FireLib User Manual and Technical Reference*, October 1996. [Online]. Available: <http://www.fire.org/downloads/fireLib/1.0.4/doc.html>
- [8] Systems for environmental management. public domain software for the wildland fire community. [Online]. Available: <http://www.fire.org>
- [9] R. C. Rothermel, "How to predict the spread and intensity of forest and range," Ogden, Utah, Dpto. de Agricultura de Estados Unidos, Servicio Forestal, Intermountain Forest and Range Experiment Station., Tech. Rep. INT-143, 1983.
- [10] J. R. Koza, *Genetic Programming. On the programming of computers by means of natural selection*. The MIT Press, 1992, massachusetts Institute of Technology. Cambridge, Massachusetts 02142.
- [11] Mpe open graphics. [Online]. Available: http://www-unix.mcs.anl.gov/mpi/www4/MPE_Open_graphics.html

Combining Scalability and Efficiency for 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—A huge challenge that parallel computing wants to overcome is to improve the performance of many MPI applications. However, some of these applications do not scale when the problem size is fixed and the number of core is increased. This scalability problem is increased when is used a hierarchical communications architecture how is included on multicore clusters. Therefore, this work presents a novel method developed for SPMD (Single Program Multiple Data) applications, which is based on finding the maximum strong scalability point while the efficiency is maintained over a defined threshold. This method integrates four phases: a characterization, a tiles distribution model, a mapping strategy, and a scheduling policy. Also, this method is focused on SPMD applications designed to use MPI libraries with high communication volumes. Our methodology has been tested with different SPMD scientific applications and we observed that the maximum speedup and scalability were located close to the values calculated with our model.

Keywords: multicore, SPMD, Performance, Scalability

1. Introduction

Nowadays, the scientific applications are developed with more complexity and accuracy and these precisions need high computational resources to be executed faster and efficiently. Also, the current trend in high performance computing (HPC) is to find clusters composed of multicore nodes as can be evidenced in the top500 list (rank of the parallel machines used for HPC). The integration of these nodes in HPC has allowed the inclusion of more parallelism within nodes. However, this parallelism must deal with some problems such as: number of cores per chip, shared cache, bus interconnection, memory bandwidth, etc.[1]. These issues are becoming more important in order to manage the application scalability and efficiency.

Also, the hierarchical communication architecture integrated on multicore clusters creates a heterogeneous environment, which affects some performance metrics such as efficiency, speedup and applications' scalability due to

the different speeds and bandwidths of each communication paths(Fig. 1), which may cause degradations in the application performance [2].

Despite of these communication issues and in order to benefit from such computational multicore cluster capacities, we focused on improving the performance application in these environments. This work is focused on calculating the maximum number of cores that maintain the strong application scalability while the efficiency is over a defined threshold. The objective of strong scalability is to maintain the problem size constant while the number of processors increases [3].

To obtain this goal, we have to consider the parallel programming paradigm, which the application has been designed, e.g., a master/worker, pipeline, SPMD, etc. Each of these paradigms has different communication patterns, which can affect the applications performance. In this sense, we consider the parallel applications designed using message passing interface (MPI) for communication and SPMD as a parallel paradigm. The SPMD paradigm was selected due to its behavior which is to execute the same program in all processes but with a different set of tiles. These tiles have to exchange their information in each iteration and these can become in a huge issue when we use a multicore environment.

The figure 1 shows an SPMD application execution over a multicore cluster. The tiles are computed in a similar time due to the homogeneity of the core. However, the communications are performed by different links with the objective of maintaining the communication pattern and each pathscan include up to one and a half order of magnitude of difference in latency for the same communication volume. These differences are translated into inefficiency that decreases the performance and do not allow us to obtain a linear strong scalability.

To solve these inefficiencies, we have developed a method that manages the communication latencies using some characteristics of each SPMD application (e.g. computation and communication tile ratio) and allows us to determine a relationship between scalability and efficiency. To achieve this performance relationship, our methodology is organized in four phases: characterization, tiles distribution model, mapping strategy, and scheduling policy, which allow us to

* This research has been supported by the MEC-MICINN Spain under contract TIN2007-64974

*Contact Autor: R. Muresano, rmuresano@caos.uab.es

†This paper is addressed to the PDPTA conference.

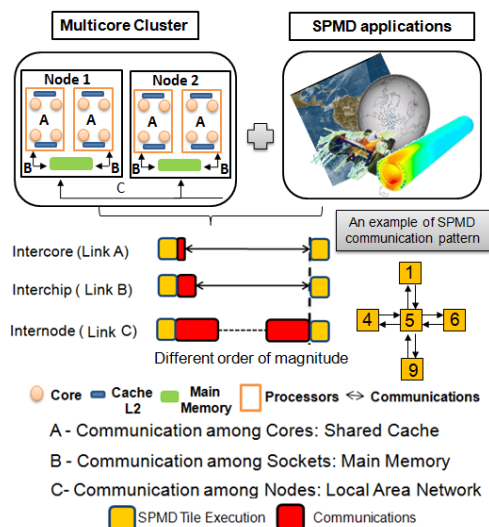


Fig. 1: Issues of SPMD applications on a multicore cluster.

distribute the tile inside the environment.

In this sense, this methodology classifies the SPMD tiles in groups, called Supertile (ST), assigning each one of them to one core. The tiles of these ST belong to one of two types: internal tiles which their communications are made in the same core and edge tiles, where their communications are performed with tiles allocated in other cores. This division allows us to apply an overlapping method, which permits us to execute the internal tiles while the edge communications are communicating. This division allows our method to find the ideal number of core that permits us to achieve the maximum strong scalability with a defined efficiency.

This paper is structured as follows: the related works are described in section 2. Section 3 exposes the issues of SPMD applications on a multicore architecture. A description of the methodology is presented in section 4. Section 5 describes the efficiency and scalability for SPMD application. Next, section 6 illustrates the performance evaluation. Finally, conclusions are given in section 7.

2. Related Works

There are different works developing methodologies which are focused on improving some performance metrics on multicore environments. Mercier et al [4] have designed a method to efficiently place MPI processes on multicore machines where establish an adequate placement policy to improve applications efficiency. However, this work does not include the combination of scalability that is very important when we wish to execute faster and efficiently.

On the other hand, Liebrock [5] defines a methodology for deriving a performance model for SPMD hybrid parallel applications. This work was focused on improving three specific performance: adaptability, scalability and fidelity using mapping, scheduling and synchronization overhead strategies designed for hybrid message passing and distribute

memory applications. On the contrary, our work evaluates pure MPI applications and similarly, we develop a methodology centered on mapping and scheduling strategies, and also, we include an efficient execution.

Moreover, there are works centered on studying and improving the efficiency [6] or enhancing the speedup on multicore clusters [7] separately. In contrast, we developed a methodology centering on mapping and scheduling strategies, and we search an improvement in both speedup and efficiency performance metrics on these clusters [8]. In this previous work, we have developed the methodology phases using the characterization, mapping and scheduling strategies. However, these phase only permit us to find the number of tiles that let us to obtain the maximum speedup of the SPMD application defining a desired efficiency. However, this current work searches for a combination of strong scalability and efficiency, in which we can predict the number of core that maintain the relationship between both metrics.

We are focused on using mapping and scheduling strategies top achieve our objective, In this sense, some works have developed mapping strategy for SPMD applications, which are centered on improving the application efficiency [7]. Another technique was designed by Brehm et al [9], in which the main objective was to map the application using the characteristics of the applications. Similarly, our proposed mapping maintains the efficiency using the characteristics of the machine and the application, but we add an affinity process that allows us to minimize the communication effect of the multicore environment.

Also, there are some scheduling strategies for SPMD applications [10] [11] that are based on finding the minimum execution time, which is part of our objective. Nevertheless, we analyzed and evaluated the model defined by Panshenkov et al [12] and we chose some characteristics such as: tiles are divided into blocks, asynchronous communications, computation and communication overlapping, with the aim of minimizing the communications overhead and improving the efficiency of the SPMD application.

3. SPMD applications on multicore

In this study, the SPMD applications used have to accomplish the following characteristics: static, where parallel application defines the communication process and is maintained during all the execution, local, where applications do not have collective communications, 2D grid applications, and regular, because communications are repeated for several iterations. In this sense, there are some benchmark that use these characteristic: one of them is NAS parallel benchmarks in the CG, BT algorithms [13] and also there are real applications such as: heat transfer simulation, Laplace equation, applications focus on fluid dynamics field like mpbl suite [14], application of finite differences etc.

Also, the communication pattern can vary according to the objective of the SPMD application. However, these patterns

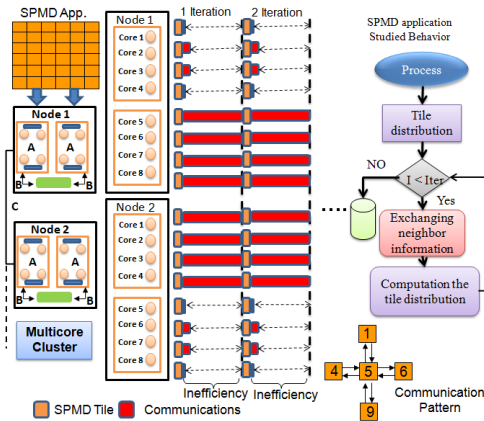


Fig. 2: SPMD application on multicore cluster.

are defined in the beginning of the application and are kept until the application end. The figure 2 shows an example of SPMD applications and multicore clusters, in which is illustrated the idle time generated by slower communication links, (e.g. cores 5-8 communicating from node 1 with core 1-4 of node 2 through the inter-node link). These Internode communications have the bigger delay that can generate huge influences in the efficiency and scalability.

However, these idle times allows us to establish strategies in order to organize how SPMD tiles could be distributed on multicore cluster with the aim of managing these communications inefficiency. These communication links can vary the communication time in an one and a half order of magnitude according to the path which perform the communication. These variations are a limiting factor to improve application performance, due to the latency of the slower link, which determines when an iteration has been completed (Fig. 2). These inefficiencies have to managed if we wish to executed the SPMD application faster, efficient and scalable.

To manage this communication issues, we use the problem size of the SPMD applications that is composed by a number of tiles and we create the SuperTile (ST). 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 searching the relationship between strong scalability and efficiency. The ST is calculated maintaining the focus of

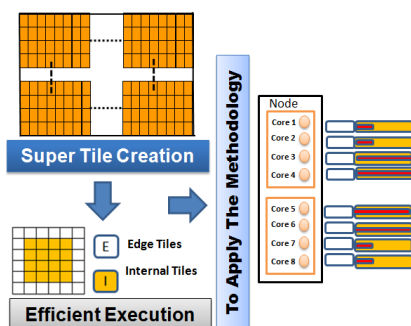


Fig. 3: SuperTile (ST) creation for improving the efficiency.

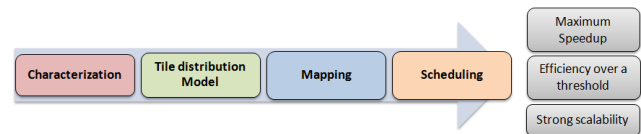


Fig. 4: Phases for efficient execution of SPMD appl.

obtaining the maximum strong scalability point while the efficiency is maintained over a defined threshold.

The figure 3 shows an example of the overlapping process and the ST creation. This ST is a group of tiles of the global problem size which is defined by $M \times M$. In this sense, this ST is integrated of a set of $K \times K$ Tiles, where K is the square root of the number of tiles, which have to be assigned to each core with the aim of maintaining an ideal relationship between efficiency, strong scalability and speedup. As mentioned before, the ST is composed by two type of tile internal and edge tile. This is done with the objective of creating an overlapping strategy that minimize the communication effects in the parallel execution time.

4. Methodology definition

This methodology is focused on managing the different communication latencies and bandwidths presents on multicore clusters with the objective of improving both efficiency and application scalability. This process is realized through four phases: a characterization, a tile distribution model, a mapping strategy and a scheduling policy (Fig. 4). These phase allow us to handle the latencies and the imbalances created due to the different communication paths.

Thus, our methodology realizes an application and environment analysis in the characterization phase with the aim of obtaining the application parameters and the computation and communication ratio which will be used to calculate the number of tile of the ST and the ideal number of cores. The next step is to calculate the tiles distribution which determines the number of tiles that have to be assigned to each core in order to achieve our objective, and also we calculate the number of core necessary to maintain both strong scalability and efficiency conditions. Next, the mapping phase allocates the set of tiles (ST) among the cores which are calculated with the model defined in the tile distribution phase. Finally, the scheduling phase has two functions, one of them is to assign tile priorities and the other is to control the overlapping process. Once the methodology is applied, we evaluate the performance results obtained.

4.1 Characterization phase

The objective of this phase is to gather the necessary parameters of SPMD applications and environment. This characterization parameters are classified in three groups: the application parameters, parallel environment characteristics and the defined efficiency. All these parameters give us the nearest relationship between the machine and the application.

The application parameters offer the information necessary of the application characteristics such as: problem size, number of tile, iteration number, communication volume and computation time of a tile, etc. Also, these parameters allow us to determine the communication pattern of a tile and determine the distribution schemes of the SPMD application.

The parallel environment parameters enable us to determine the communication and computational time of a tile inside the hierarchical communication architecture. These values of a tile obtained allow us to calculate a ratio between them. This ratio will be defined as $\lambda_{(p)(w)}$, where p determine the paths where a tile has to communicate with neighboring tile, e.g. through A, B, or C link (Fig. 1). The variable w describes the direction of the communication e.g. up, right, left or down in a four communications pattern. This ratio is calculated with equation 1, where $Comt_{(p)(w)}$ determines the time of communicating a tile for a p paths and the Cpt is the computing time of a tile.

$$\lambda_{(p)(w)} = Comt_{(p)(w)} / Cpt \quad (1)$$

Finally, once all parameters are found through the characterization phase, we include the efficiency value in the model and evaluate the execution time. The efficiency value is defined by the variable $effic$ and this will be included in the model.

4.2 Tile distribution model phase

The main objective of this model is to determine the number of cores $Ncores$ that allow us to maintain the relationship between the maximum strong scalability and the desired efficiency. In this sense, equation 2 calculates the number of core. This equation depends on the problem size which is represented by the M^2 divided by the optimal number of tile K^2 (Equ. 2).

$$Ncores = M^2 / K^2 \quad (2)$$

Knowing the value of K , we can estimate the execution time of the SPMD application. For example, the equation 3 represents the behavior of SPMD application using the overlapping strategy, where first is calculated the edge tile computation ($EdgeComp_i$) and then is added with the maximum value between internal tile computation ($IntComp_i$) and edge tile communication ($Edgcomm_i$). This process will be repeated for a set of iterations ($iter$) where n determine the number of an iteration. This process is carried out for the communication exchanging of the SPMD application and can be possible to calculate due to the deterministic behavior of these applications.

$$Tex_i = \sum_{n=1}^{iter} (EdgComp_{(i)} + Max \left\{ \begin{array}{l} IntComp_{(i)} \\ Edgcomm_{(i)} \end{array} \right\}) \quad (3)$$

$$EdgeComp_{(i)} = 4 * (K - 1) * Cpt \quad (4)$$

$$IntComp_{(i)} = (K - 2)^2 * Cpt \quad (5)$$

$$Edgcomm_{(i)} = K * Max(Comt_{(p)(w)}) \quad (6)$$

The edge communication (Equ.6) has to be for the worst communication case. This means that we use the slowest communication time to estimate the number of tiles necessary for maintaining the efficiency. To do this, we have to calculate the $\lambda_{(p)(w)}$ ratio (Equ 1) explained before.

Then, the first step is to determine the ideal value of K . We start from the overlapping strategy, where internal tile computation ($IntComp_{(i)}$) and the edge tile communication ($Edgcomm_{(i)}$) are overlapped. The equation 7 represents the ideal overlapping that allow us to obtain the maximum speedup while the efficiency $effic$ is maintained over a defined threshold. Therefore, we start from an initial condition where the edge communication time is bigger than the internal computation time divided by the efficiency. This division represents the maximum inefficiency allowed.

$$K * Max(Commt_{(p)(w)}) >= ((K - 2)^2 * Cpt) / effic \quad (7)$$

However, this equation 7 has to consider a constraint defined in equation 8 where $Edgcomm_{(i)}$ can be bigger than $IntComp_{(i)}$ over the defined efficiency (Equ. 7), but the $Edgcomm_{(i)}$ have to be slower than the $IntComp_{(i)}$ without any efficiency definition.

To calculate the optimal value of K , we determine the $\lambda_{(p)(w)}$ (Equ. 1) value and we solve the $Commt$ value, which can be calculated with respect to $\lambda_{(p)(w)}$ multiplied by computational time Cpt of a tile. This process is performed to equalize both internal computation and edge communication equations in function of Cpt . This value is replaced in equation 7 and we obtain the equation 9.

$$K * Max(Commt_{(p)(w)}) <= ((K - 2)^2 * Cpt) \quad (8)$$

To find the value of K , we equal the equation 9 to zero and we obtain a quadratic equation (Equ. 10). These two solutions obtained have to be replaced in equations 7 and 8, with the aim of validating if the k value accomplish the constraint defined.

$$effic * K * Cpt * Max(\lambda_{(p)(w)}) = (K - 2)^2 * Cpt \quad (9)$$

$$K^2 - (4 + effic * max(\lambda_{(p)(w)}) * K + 4 = 0 \quad (10)$$

The next step is to calculate the number ideal of core (Equ. 2), which are needs to find the strong scalability with the desired efficiency. To do this, we start of the initial consideration that establish that one ST will be assigned to each core and we use the equation 2, that calculate the ideal number of core that allow us to obtain the objective stated. This number of core determines the inflection point until the application has a strong scalability. Finally, we can determine

the theoretical behavior of the SPMD application for a lower number of cores that the optimal calculated and predict its behavior. Equation 11 calculates the new values of K for a number of core given with the objective of determining the execution time with equation 3.

$$K = \sqrt{M^2/Ncores} \tag{11}$$

4.3 Mapping strategy phase

A set of difficulties arise when we allocate SPMD tile into distinct cores and these cores have to communicate through different links. Under this focus, the main objective of this phase is to design a strategy of allocating the ST into each core with the aim of minimizing the communication effects. The ST assignments are made applying a core affinity which allows us to allocate the set of tiles according to the policy of minimizing the communications latencies [4]. This core affinity permits us to identify where the processes have to be allocated and how the ST can be assigned to each core.

The next step in this phase is to create a logical processes distribution that allows the application to identify the neighbor communications. This is done using a cartesian topology of the processes that give to each process two coordinate in the grid distribution. These two coordinates identify the cores, in which the processes have to be allocated. Also, we can coordinate the communication order with the objective of minimizing the saturation of the links. The last step is to create the ST with the values obtained with the model.

4.4 Scheduling policy phase

The scheduling phase is divided into two main parts: the first one is to develop an execution priority which determines how the tiles have to be executed inside the core and the second part of the scheduling phase which is focused on applying an overlapping strategy between internal computation and edge communication tiles.

The execution priority assignments are assigned by each tile and the highest priorities are established for tiles which have communications through slower paths. These assignments have the following policies: tiles with external communications are selected with priority 1. These edge tiles are saved in buffers with the aim of executing these tiles first. These buffers are updated all iterations. The second assignation is made for internal tiles which are overlapped

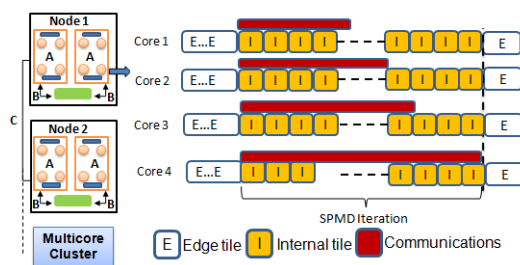


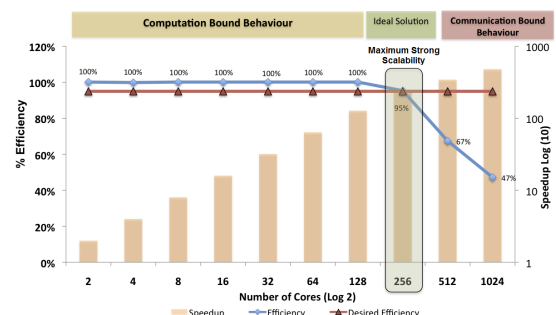
Fig. 5: Scheduling policy.

with the edge communications, which are assigned with the priority 2. The overlapping process uses two threads, one of them is to perform the internal computation and the other is to manage the asynchronous communications. These communications enable us to perform the internal computation and the edge communication together (Fig. 5).

5. Combining scalability and efficiency

Our methodology attempts to find the number of core that achieves the maximum strong scalability with a defined efficiency. However, there are two distinct definition of scalability in HPC. One definition is the weak scalability that is considered when the problem size and the number of processing elements are expanded. The main goal of this scalability is to achieve constant time-to-solution for larger problems and the computational load per processor stays constant [3]. The second definition is the strong scalability in which the problem size is fixed and the number of processing elements is increased. The goal in this scalability is to minimize the time solution. Hence, the scalability means that speedup is roughly proportional to the number of processing elements.

Under these two scalability definition, our methodology searches a combination between strong scalability and efficiency. This combination means that our analytical model has to determine the number of cores that allow us to obtain the ideal relationship between speedup and the defined efficiency. This number of cores can be calculated using the model and this number determines the maximum systems capacity growth. Also, we can determine the theoretical behavior of the application (Equ.2. This equation allows us to find the K value size that have to be assigned to each core. The model only finds one ideal value to maintain the ideal overlapping. However, we can calculate values for another number of cores with the aim of evaluating the performance.



Cores	K	Edge Cp	Int Cp	Edge Comm	Exec T
16	396	1580	155236	39600	156816
32	280	1176	77284	28000	78400
64	198	788	38416	19800	39204
128	140	556	19044	14000	19600
(256)	99	392	9409	9900	10292
512	70	276	4624	7000	7276

Fig. 6: Combining scalability and efficiency of SPMD appl.

5.1 A theoretical Example

This numerical example illustrates how we can combine the efficiency and the strong scalability concept. Suppose the following application characteristics: a defined problem size of $M=1585$, a defined efficiency (*Effic*) of 95%, a four communication pattern, three different communication links (e.g. cache, main memory and network) and a set of node of double quad core architecture. Then, we have to determine the $\lambda_{(p)(w)}$ using equation 1 and we have to use the maximum value obtained. We assume that the $C_{pt} = 1$ time unit and the maximum communication time for the slowest communication paths $Comm_t = 100$ time units.

Afterward, we apply our analytical model, where we use the equation 10 to determine the ideal ST and the equation 2 to calculate the ideal number of cores, which represents the maximum combining strong scalability and efficiency. The ideal value of K obtained is around 99 and the ideal number of core is equal to 256 for this example (Eq. 2). Once the K and N_{cores} are calculated, we have to determined the efficiency and speedup for this ideal number of core. In this sense, we have to calculate the serial execution time using the global problem size multiplied by one computational tile time C_{pt} . This example is for one iteration and it has a serial time of 2.512.225 time units for this specific problem size.

The figure 6 shows the result for a different distribution of cores with the aim of visualizing the efficiency and speedup curve for this example. Also, it illustrates the performance behavior for different number of cores, in which the ideal number calculated have the efficiency around the optimal value defined and the speedup until this point has a roughly linear growth. This point is the maximum strong application scalability under a desired efficiency. After this ideal point, we can observe that speedup increases but no proportionally to the number of core and the efficiency begins to decrease considerably due to the communication bound behavior. On the contrary, before the ideal point the efficiency and speedup are around the maximum values (Fig. 6).

6. Performance Evaluation

The experiments to test our methodology has been conducted on two multicore clusters, one of them is a DELL cluster with 8 nodes with 2 Quadcore Intel Xeon E5430 of 2.66 Ghz, 6 MB of cache L2 shared by each two core and 12 GB of RAM memory. The second is an IBM with 32 nodes with 2 processors Dual-Core Intel(R) Xeon(R) CPU 5150, 4 MB shared cache memory by 2 cores, 12 GB of RAM and gigaethernet network. Both clusters have Openmpi 1.4.1. To validate the result of this article, we chose two applications: heat transfer and one application of fluid dynamics (LL-2D-STD-MPI) integrated in the MP-Labs suite.

6.1 Efficiency and scalability evaluation

The main objective of this evaluation is to demonstrate the improvement of applying our methodology. The table 1

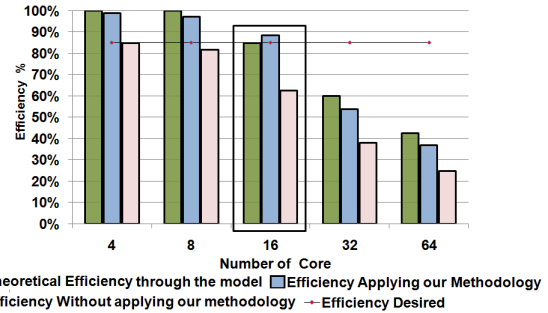


Fig. 7: Efficiency of Heat Transfer App. on Dell Cluster

shows the characterization values of computation (C_{pt}) and $Comm_t_{(p)(w)}$ of slowest communication of a tile, problem size, desired efficiency and also illustrates the theoretical values of number of cores, edge and internal computation, the edge communication and the number of tiles.

To develop our performance analysis, we executed SPMD applications but making a comparison between the theoretical value, the application without using and the application using our methodology. In this sense, the figure 7 shows the efficiency behavior of heat transfer application executed with 100 iterations. This figure 7 illustrates a considerable improvement in efficiency of around 42% when we execute with the number of cores determined by our model. Also, we can observe how the application using our methodology behaves similarly to the analytical model where the error rate is around 5% when the number of core is below to the maximum obtained of our model (Fig. 7).

The figure 8 shows how the speedup increases when we add more core, but this speedup do not scale linearly after the maximum number of core determined with our model. The ideal point calculated meets the maximum strong scalability while the efficiency is over a defined threshold.

On the other hand, the LL-2D-STD-MPI application is integrated by 3 main parts: prestep, poststep, and the main module where the communication and the computation is performed. In this order, we apply our methodology and the tile characterization process to the last module, because the other two only compute and they do not have any communication and this application has been tested with 100 iterations. The figure 9 shows the improvement in the efficiency between the original version and when we applied our methodology with an error rate of 4% of precision

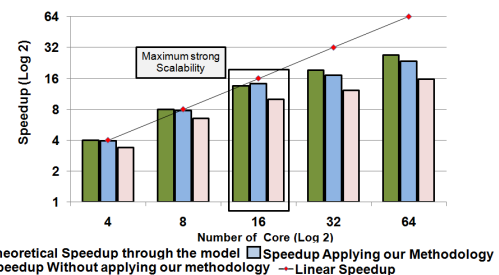


Fig. 8: Speedup of Heat Transfer App. on Dell Cluster

Table 1: Tile distribution model evaluation examples

App.	C_{pt}	$Comt_{p,w}$	M	effic	K	$IntComp$	$EdgeComp$	$EdgeComm$	Ncore	Cluster
Heat Tran	$0.021\mu sec$	$58, 8\mu sec$	9500×9500	85%	2384	$1, 18E-01Sec$	$1, 99E-04Sec$	$1, 40E-01Sec$	16	DELL
LL-2D-STD	$0.24\mu sec$	$60, 7\mu sec$	2000×2000	95%	249	$1, 458E-02Sec$	$2, 34E-04Sec$	$1, 48E-02Sec$	64	IBM

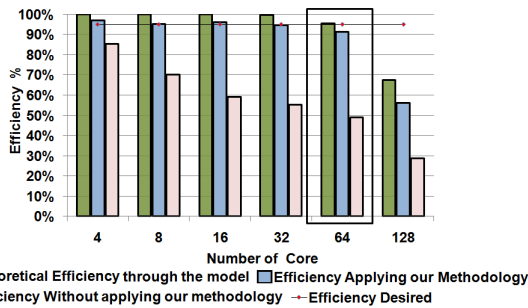


Fig. 9: Efficiency of LL-2D-STD-MPI app on IBM Cluster

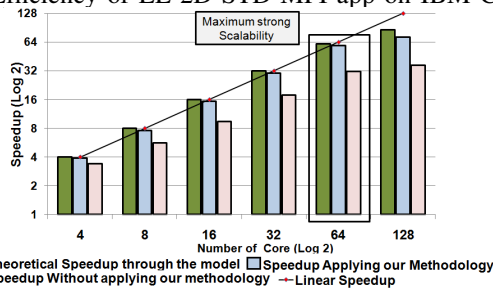


Fig. 10: Speedup of LL-2D-STD-MPI app on IBM Cluster with the analytical model. Similarly to the heat transfer application, the model gives us the ideal ST and number of core that maintain the relationship between efficiency and scalability (Table 1).

Finally, we observe the behavior of speedup and the strong scalability in this application in figure 10, where we can observe the linear speedup until the number of cores is below to the theoretical value. This allows us to conclude that our methodology can determine the maximum strong scalability combining the maximum speedup and the efficiency over a defined threshold. The objective to test the application in two clusters is due to check the functionality of our method in different multicore architectures. These two examples show an approximation of our method and how the maximum speedup is reached with the efficiency defined in the model.

7. Conclusion and Future Work

This work addresses how we can combine the efficiency and the strong scalability in parallel applications. Also, it was presented a novel methodology based on characterization, a tile distribution model, a mapping strategy and a scheduling policy. These phases allowed us to find through an analytical model the optimal size of the Supertile and the number of core needs to accomplish the objective stated. This model is focused on managing the hierarchical communication architecture presents on multicore clusters.

The experimentations have demonstrated that this optimal size can achieve the conditions of maximum speedup and

efficiency over a defined threshold. To achieve this, we have proposed an appropriate manner to manage the inefficiencies generated by communications links presented on multicore clusters, as was described. In addition, with our method we can observe how the SPMD applications with some specific characteristics can behave with a specific problem size while is incremented the number of cores. This is the main purpose of finding the maximum point that allows the SPMD application to scale linearly. Future works are focused on working with heterogeneous computation on multicore environment with the aim of executing the SPMD applications efficiently in an communication and computation heterogeneous environments.

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] M. Mccool, "Scalable programming models for massively multicore processors," *Proc. of the IEEE*, vol. 96, no. 5, pp. 816–831, 2008.
- [3] L. Peng, M. Kunaseth, H. Dursun, K. ichi Nomura, W. Wang, R. K. Kalia, A. Nakano, and P. Vashisht, "A scalable hierarchical parallelization framework for molecular dynamics simulation on multicore clusters," *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications, USA*, pp. 97–103, 2009.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] K. Vikram and V. Vasudevan, "Mapping data-parallel tasks onto partially reconfigurable hybrid processor architectures," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 14, no. 9, p. 1010, 2006.
- [8] 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).
- [9] J. Brehm, P. H. Worley, and M. Madhukar, "Performance modeling for spmd message-passing programs," *Concurrency - Practice and Experience*, vol. 10, no. 5, pp. 333–357, 1998.
- [10] O. Beaumont, A. Legrand, and Y. Robert, "Optimal algorithms for scheduling divisible workloads on heterogeneous systems," *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, p. 98, 2003.
- [11] J. B. Weissman and X. Zhao, "Scheduling parallel applications in distributed networks," *Cluster Computing*, vol. 1, pp. 109–118, 1998.
- [12] M. Panshenskov and A. Vakhitov, "Adaptive scheduling of parallel computations for spmd tasks," *ICCSA 2007*, pp. 38–50, 2007.
- [13] 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.
- [14] T. Lee and C.-L. Lin, "A stable discretization of the lattice boltzmann equation for simulation of incompressible two-phase flows at high density ratio," *J. Comput. Phys.*, vol. 206, pp. 16–47, June 2005.

A Methodology to Calculate a Program's Robustness against Transient Faults

Joao Gramacho, Dolores Rexachs, Emilio Luque

Computer Architecture and Operating Systems Department
Universitat Autònoma de Barcelona
Bellaterra (Barcelona), Spain

Abstract - *Computer chips implementation technologies are evolving to obtain more performance. The side effect of such a scenario is that processors are less robust than ever against transient faults. As on-chip solutions are expensive or tend to degrade processor performance, the efforts to deal with these transient faults in higher levels are increasing. Software based fault tolerance approaches against transient faults often use fault injection experiments to evaluate the behavior of applications with and without their fault detection or fault tolerance proposals. Those fault injection experiments consumes lots of CPU time by running or simulating the application being evaluated as many times as necessary to obtain a reasonable valid statistical approximation. This paper proposes the concept of a program's robustness against transient faults and presents a methodology for exhaustively calculate this robustness based on program's execution trace over an architecture and on information about the used architecture. The presented approach, besides calculating a precise robustness, accomplishes its work faster than using fault injection experiments with $\pm 2\%$ of confidence interval.*

Keywords: transient faults, robustness, reliability, swifi.

1 Introduction

The ever growing die density of computer processors is one of the great factors of the astonishing improvements in processing power of the last decades. Computer chips are using smaller components, having more transistors, using those transistors with higher density and also operating at lower voltage. The side effect of such a scenario is that processors are less robust than ever against transient faults [1].

Transient faults are those faults that might occur only once in a system lifetime and never happen again the same way. Transient faults in computer systems may occur in processors, memory, internal buses and devices, often resulting in an inversion of a bit state (i.e. single bit flip) on the faulty location [2]. Cosmic radiation, high operating temperature and variations in the power supply subsystem are the most common cause of transient faults in computer systems.

A transient fault may cause an application to misbehave (e.g. write into an invalid memory position; attempt to

execute an invalid instruction). Such misbehaved applications will then be abruptly interrupted by the operating system fail-stop mechanism. Nevertheless, an undetected data corruption is the biggest risk for applications. It happens when the flipped bit produced by the transient fault generates an incorrect final program result that might not be ever noticed. Errors that can be noticed as effects of transient faults are called soft errors.

In order to test a program behavior in presence of transient faults, it is common to put the program to be tested in an environment designed to allow transient like fault injections. In this way, it is possible to evaluate if the program misbehaved in presence of a transient fault or if the program was robust and could finish properly or could detect the injected fault and stopped its execution avoiding the error propagation. These fault injections are made often by flipping a bit of a processor register in a given point during program execution.

In this work we propose the concept of robustness against transient faults as the ability of a program, once in presence of a transient fault, to keep running and give a correct result when finish or to stop the execution when a soft error is detected and inform about it.

We consider that a program running over an determined architecture will have a robustness against transient faults represented as a number that can vary from zero (0%) to one (100%), where zero implies no robustness at all (the program fail on every possible cases) and one implies the best robustness possible (the program gave the correct result or has detected the transient fault on every possible cases).

Execute a program with fault injections to evaluate its behavior can be a time consuming task. This is because of the need to execute the program in the fault injection environment as many times as needed to have a result with significant statistical approximation, as we will show in section 2.

Our objective in this work is to propose a methodology to calculate a program's robustness against transient faults without any fault injection execution.

To do so, in section 3 we present how to calculate the robustness of a program running over a given architecture based on a trace of the program execution over the architecture and also on information about how the architecture instructions deal with processor registers. Our

methodology obtains a precisely calculated robustness avoiding the time consuming task of doing hundreds (or thousands) of program executions in transient fault injection environments.

In section 4 we present an experimental evaluation of the proposed methodology comparing the result of a set of fault injection campaigns with the result obtained using methodology and in section 5 we present our conclusion and explain about the next steps of our work.

2 Evaluation Using Fault Injection

Experimental methods of injecting transient faults into a program during its execution were proposed to test purposed protection mechanisms against transient faults. On those methods, the program being evaluated is executed in an environment able to inject a fault in a form of a bit flip on a program architectural state (usually a bit in a processor register). At the end of the program execution, its result is evaluated to check the effect caused by the fault into the execution.

When the program finished correctly and presented the same result of a fault free execution the program architectural bit changed by the fault injection is classified as unACE (unnecessary for an Architecturally Correct Execution). On the other hand, when the program didn't finished correctly, or presented a result different of the fault free execution, the program architectural bit changed by the fault injection is classified as ACE (necessary for an Architecturally Correct Execution).

If the program being evaluated has some kind of fault detection mechanism against transient faults the program architectural bits changed may trigger the fault detection mechanism and lead the program to a fail stop avoiding the propagation of the fault effect in the program execution. On those cases, instead of being classified as ACE, as the execution finished doing a fail stop and noticed that a fault happened the program architectural bit changed is classified as DUE (Detected Unrecoverable Error).

As changes in the ACE program architectural bits lead to an abnormal program behavior and also could lead to a result different of the obtained by a fault-free execution, it is common to classify those bits as SDC (Silent Data Corruption).

$$\frac{\text{unACE bits} + \text{DUE bits} + \text{SDC bits}}{\text{amount of tested bits}} = 1 \quad (1)$$

To evaluate how reliable a program is in presence of transient faults with a sufficient large amount of executions with fault injection, we can divide the amount of executions that didn't failed (those in which the program architectural bit changed was classified as unACE or DUE) by total amount of executions with fault injection performed. Also, it is important to have a good distribution in which program architectural bit is changed on each execution, since it is randomly chosen.

The authors of [3] propose a soft error detection mechanism based on source code transformation rules. The

new program (compiled with the source code transformed with the fault detection mechanism) has the same functionality as the original program but is able to detect bit-flips in memory and processor registers during an execution.

Evaluating programs with and without their fault detection mechanism, the authors of [3] performed a set of fault injection experiments where on each execution a bit was flipped in processor registers, program code memory region or program data memory region. A total of 52,728 executions with fault injection were performed to evaluate two programs (the original one and the changed to detect soft errors), 26,364 executions per program on average.

In Error Detection by Duplicated Instructions (EDDI) [4], the authors reduced the amount of SDC cases of programs by, during program's compilation, copying instructions but using different processor registers and adding verification for errors by comparing the value of the original processor register used by the program with the value of the processor register used in the new generated instruction.

Executing a total of four evaluations (the original program, the program with EDDI and the program with three source code based fault detection mechanisms) per each of the eight benchmarks evaluated and executing 500 simulations with fault injection per evaluation, the authors of [4] have made a total of 16,000 simulations to accomplish their work.

On Software-Controlled Fault Tolerance [5], the authors presented a set of transient fault detection techniques based on software and also hybrid (based on software and hardware). Each of the proposed techniques has a different cost/benefit relation by improving reliability or performance.

The first technique presented by [5] is SWIFT (Software Implemented Fault Tolerance) which reduces an application's amount of SDC cases by changing the program during compilation time. The other techniques presented are all hybrid. The set of those hybrid techniques is called CRAFT (Compiler-Assisted Fault Tolerance). In general, reduces the amount of SDC cases even more than SWIFT and also improve the performance of the program in comparison with software-only fault tolerance techniques.

To evaluate the amount of SDC cases of an application with and without the proposed fault tolerance mechanisms, the authors of [5] executed fault injection experiments in a simulator executing all programs to the end using a functional simulator and choosing when and where to inject the fault randomly. The authors classified the fault injection simulation result as unACE if the flipped bit wasn't necessary to the correct architectural execution, as DUE if the flipped bit triggered a fault detection mechanism, or as SDC if the flipped bit generated a silence data corruption.

The authors of [5] used a benchmark to evaluate how many fault injections should be necessary to have a significant statistical approximation of results. They executed 5,000 fault injection simulations with the benchmark and observed that the confidence interval of the average of the

SDC cases was $\pm 2.0\%$ after 946 simulations, $\pm 1.5\%$ after 1,650 simulations and $\pm 1.0\%$ after 3,875 simulations.

In a total of 10 sets of experiments, the authors of [5] evaluated the robustness of a set of benchmarks by simulating 5,000 executions with fault injection (except for two SWIFT variations that used 1,000 simulations). In each of 504,000 simulated executions with fault injection a randomly chosen bit of one of 127 integer processor registers of IA64 processor architecture was flipped.

Because of the use of a simulator to execute the program with a fault injection, the authors of [5] could save some simulation time on the executions where the bit flipped was classified as unACE. On those cases, the simulation could be interrupted when the simulator observed that the flipped bit was re-written with results from processor logical unit or with a write operation before having it content used.

Continuing their research in fault tolerance for transient faults, the same authors of [5] propose Spot [6], a technique to dynamically insert redundant instructions to detect errors generated by transient faults. This dynamically insertion was made in runtime using instrumentation.

Besides using a different architecture from previous work (in [6] they used IA32 and protected only the eight general purpose 32 bit registers of the architecture), the authors didn't use simulators. All the analysis and fault injections were made using an instrumentation tool. The authors of [6] evaluated 16 benchmarks and executed a total of 1.03 million fault injections to obtain their results (keeping 5,000 executions with fault injection per benchmark and configuration evaluated).

In all related work studied, the execution of a program in a transient fault injection environment could be classified in terms of basically three labels: unACE, DUE and SDC.

To compute a program's robustness against transient faults using fault injection we only need to divide the amount of unACE cases, plus the amount of DUE cases, by the amount of executions made in the experiment.

$$robustness = \frac{unACE\ bits + DUE\ bits}{amount\ of\ tested\ bits} = 1 - \frac{SDC\ bits}{amount\ of\ tested\ bits} \quad (2)$$

If all executions are classified as SDC, the robustness will be zero (the minimal robustness allowed). On the other hand, if all executions are classified as unACE or DUE, the robustness will be one (the maximum robustness allowed).

The robustness evaluation method using program executions with fault injection need a sufficient large amount of executions varying the fault conditions (time, register and bit) to have a representative statistical approximation of the results.

One aspect that must be took into account when using fault injection to evaluate a program's robustness is that this method is data dependent. Faults injected in specific bits of floating point registers can lead to almost no change in its value depending on its original value. Also, as general purpose registers are often used as pointers to vectors or matrices of data, if this data is homogenous (e.g. a vector

filled with ones) there are many changes that can be done in registers that will make them point to a different memory position but with the same data, masking the fault injection result as unACE.

Also, it is known that by using a fault injection based evaluation of robustness, the amount of executions to evaluate a program will affect the precision of robustness obtained [5].

Finally, using simulators or dynamically instrumentation to inject fault on every program execution will increase time needed on each execution in comparison with a time spent by the program running directly in the architecture without instrumentation.

3 The Proposed Methodology

The objective of this paper is to present a methodology of exhaustively calculate the amount of bits that we can classify as unACE or DUE of a program execution (and, so, its robustness against transient faults) without using executions with fault injection.

In this step of our methodology we are only classifying the unACE bits of a program state. So, the robustness that we will calculate using the presented methodology will be relative only to those architectural state bits that for sure don't affect the program final result.

As the programs used in our experimental evaluation don't have any type of added protection against transient faults we will assume that the amount of bits that could be classified as DUE are zero and that the program's robustness rely only in the evaluation of the unACE bits.

$$robustness = \frac{unACE\ bits}{amount\ of\ tested\ bits} \quad (3)$$

The robustness that we are going to calculate using our methodology is relative to a program *prog* running over a determined architecture *A*.

$$robustness_{prog \times A} = \frac{unACE\ bits}{amount\ of\ tested\ bits} \quad (4)$$

We will need two things in order to know how many architectural state bits are in an evaluation (the amount of tested bits), one from the architecture and other from the program. From the architecture we will need the amount of bits that could be changed by a fault during each processor instruction executed by the program. From the program, we will need the amount of instructions it executes to produce its results.

$$amount\ of\ tested\ bits = tested\ bits\ per\ instr \times amount\ of\ instr \quad (5)$$

As our methodology is based on Software Implemented Fault Injection (SWIFI) methods, the amount of bits that can be changed by a fault during each processor instruction executed by the program can be easily calculated by summing all processor register's size. We now define a set named $ProcReg_A$ with all processor registers that we will consider in our evaluation and a non-numerical finite sequence $RegSize_A$ representing the size in bits of each processor register in $ProcReg_A$. The relation between $ProcReg_A$ set and $RegSize_A$ sequence is defined by the function $f_{RegSize}$.

$$ProcReg_A = \{reg_1, reg_2, \dots, reg_{nreg(A)}\} \quad (6)$$

$$f_{RegSize}: ProcReg_A \mapsto \mathbb{N}$$

$$RegSize_A = (regsize_{reg_1}, regsize_{reg_2}, \dots, regsize_{reg_{nreg(A)}})$$

$$tested\ bits\ per\ instr = \sum_{r=1}^{nreg(A)} f_{RegSize}(r)$$

In order to calculate the amount of instructions executed by the program we will need an execution trace of the program running over the defined architecture. This trace will be represented as a non-numerical finite sequence $Trace_{prog \times A}$ defined by the function f_{prog} . The f_{prog} function returns the processor instruction executed by the program in a given point of its execution. The instruction must be a member of the $ProcIns_A$ set (that contains all possible processor instructions).

$$ProcIns_A = \{ins_1, ins_2, \dots, ins_{nins(A)}\} \quad (7)$$

$$f_{prog}: \mathbb{N} \mapsto ProcIns_A$$

$$Trace_{prog \times A} = (ins_1, ins_2, \dots, ins_{nins(Trace_{prog \times A})})$$

$$amount\ of\ instr = nins(Trace_{prog \times A})$$

At this point we have defined the first part of our formula to calculate program's robustness against transient faults when running over a given architecture.

$$robustness_{prog \times A} = \frac{unACE\ bits}{nins(Trace_{prog \times A}) \times \sum_{r=1}^{nreg(A)} f_{RegSize}(r)} \quad (8)$$

3.1 Robust state

Let's consider now robust state as a property of a processor register in a given point of a program execution. This register property will be represented by a vector of logical states (true or false) with as many states as the amount of bits of the processor register, and it will be defined by the f_{rstate} function that we will explain better later.

$$f_{rstate}: ProcReg_A \times \mathbb{N} \mapsto \mathbb{B} \quad (9)$$

An element of a register robust state vector being true implies that the register bit represented by the element is classified as unACE in the given execution point of the program. In this way we know that any change in this register bit in this given execution point of the program won't be propagated to the final program result.

Similarly, an element of a register robust state vector being false implies that the register bit represented by the element is classified as ACE (we don't know yet if DUE or SDC) in the given execution point of the program. In this way we know that any change in this register bit in this given execution point of the program can be propagated to the final program result.

In order to know how many robust state vector elements are true (to know how many bits of a register robust state are classified as unACE in a given point of program execution) we will need the f_{abits} function. This function needs a logical states vector as input parameter and will return the amount of logical states of the vector that have its value as true.

$$f_{abits}: \mathbb{B} \mapsto \mathbb{N} \quad (10)$$

With the two previously presented functions we are able to complete our general robustness formula.

$$robustness_{prog \times A} = \frac{\sum_{n=1}^{nins(Trace_{prog \times A})} \sum_{r=0}^{nreg(A)} f_{abits}(f_{rstate}(r,n))}{nins(Trace_{prog \times A}) \times \sum_{r=0}^{nreg(A)} f_{RegSize}(r)} \quad (11)$$

A program's robustness against transient faults when running over a determined architecture will be the sum of the amount of bits classified as unACE of each processor register r of the given architecture A in every point n of the program $prog$ execution present on trace $Trace_{prog \times A}$, divided by the sum of the amount of bits of each processor register multiplied by the amount of instructions present in the trace $Trace_{prog \times A}$.

To define our f_{rstate} function we will use the method presented by [5] to save simulation time on those cases where the fault injection was applied in a processor register that had its value overwritten by a new one before any read of the content changed by the fault injection.

In the example presented in Figure 1 with a basic block sample of one of the programs used in our experimental evaluation, it is possible to notice that we can evaluate if a processor register's bits are important in a given point by only observing program's instruction sequence.

For example, as the at instruction in address $0x401a68$ load the $r9$ processor register with a given value, any change done in $r9$ before the execution of this instruction will be discarded. So, if a fault injection mechanism injects a fault on any of $r9$ bits between the executions of the instructions at address $0x401a40$ and $0x401a68$ the program result will not be affected and the bit changed by the fault injection will be classified as unACE.

On the other hand, after $r9$ be loaded by the instruction at address $0x401a40$, as the loaded value will be used by the instruction at address $0x401a84$, the $r9$ integrity must be kept in the interval between the load (write operation on register) and the use of the loaded value (read operation on register).

As the presented basic block represent a loop, while the program execution stay in the loop, the next instruction that will manipulate $r9$ after the execution of the instruction in address $0x401a84$ will be one that will change its value (a write operation on $r9$, exactly the instruction at address $0x401a68$) and, so, the integrity of the register value in this interval is no longer needed anymore.

By the previously analyzed situation we can assume that, once knowing that a register will have its value replaced by a new one (after a write operation on the register) the registers bits can be classified as unACE on every instruction executed before the one with the write operation, until an instruction that read the content of the register be found.

Address	Instruction	Register Use	unACE
0x401a40	mov r13, 0x3f00000000000000	write on r13	r13, r12, r11, r10, r9
0x401a44	mov r12, 0x3f00000000000000	write on r12	r12, r11, r10, r9
0x401a54	mov r11, 0x3f00000000000000	write on r11	r11, r10, r9
0x401a5e	mov r10, 0x3f00000000000000	write on r10	r10, r9
0x401a68	mov r9, 0x3f00000000000000	write on r9	r9
0x401a72	add ecx, 0x1	read and write on ecx	
0x401a75	mov qword ptr [rdx], r13	read on rdx and r13	
0x401a78	mov qword ptr [rdx+0x8], r12	read on rdx and r12	r13
0x401a7c	mov qword ptr [rdx+0x10], r11	read on rdx and r11	r13, r12
0x401a80	mov qword ptr [rdx+0x18], r10	read on rdx and r10	r13, r12, r11
0x401a84	mov qword ptr [rdx+0x20], r9	read on rdx and r9	r13, r12, r11, r10
0x401a88	add rdx, 0x28	read and write on rdx	r13, r12, r11, r10, r9
0x401a8c	cmp ecx, ebx	read on ecx and ebx	r13, r12, r11, r10, r9
0x401a8e	jnz 0x401a40		r13, r12, r11, r10, r9

Figure 1 – A sample basic block code.

The easiest way to analyze the execution of a program in search of those relations between uses of processor registers in read or write operations is by looking the program execution trace $Trace_{prog \times A}$ in reverse order, beginning by the last executed program instruction and following the trace until the first program instruction executed.

In this way, every time we find an instruction that write content to a processor register we can turn the logical states of the register's robust state vector elements to true (classify as unACE) until find an instruction that read the register content.

On the other hand, every time we find an instruction that read content from a processor register we can turn the logical states of the register's robust state vector elements to false (classify as ACE) until find an instruction that write content on the register.

If a given processor instruction operates a register for both read and write (e.g. as an increment operation), as our analysis is done in program trace instructions backwards, we first evaluate the write operation and then the read operation.

In our methodology we also need to know how each processor instruction deals with processor register bits for read and write. So, we will use a set named $ProcsReg_A$, which contains all ordered pairs of a processor instruction combined with a processor register.

$$ProcsReg_A = \{(ins_1, reg_1), \dots, (ins_{nins(A)}, reg_{nreg(A)})\} \quad (12)$$

For each pair in $ProcsReg_A$ set we must have an element in two non-numerical sequences: $WrittenBits$, defined by the f_{wbits} function and $ReadBits$, defined by the f_{rbits} function.

The f_{wbits} function returns a vector of logical states with all states that represent processor register bits written by the instruction with true as value.

$$f_{wbits}: ProcsReg_A \times ProcReg_A \mapsto \mathbb{B} \quad (13)$$

The f_{rbits} function returns a vector of logical states with all states that represent processor register bits read by the instruction with true as value.

$$f_{rbits}: ProcsReg_A \times ProcReg_A \mapsto \mathbb{B} \quad (14)$$

Knowing how a processor instruction operated a given processor register for read and write, and also because our analysis is done by evaluating a program trace backwards, by the truth table presented in Table I we deduced a formula to the f_{rstate} of a given processor register in a given point of program trace.

Table I – Truth table of the f_{rstate} function.

Previous Robust State	f_{wbits}	f_{rbits}	New Robust State	Description
TRUE	TRUE	TRUE	FALSE	Wasn't important, write on it, read from it, change to being important
TRUE	TRUE	FALSE	TRUE	Wasn't important, write on it, keep don't being important
TRUE	FALSE	TRUE	FALSE	Wasn't important, read from it, change to being important
TRUE	FALSE	FALSE	TRUE	Wasn't important, didn't operate, keep don't being important
FALSE	TRUE	TRUE	FALSE	Was important, write on it, read from it, keep being important
FALSE	TRUE	FALSE	TRUE	Was important, write on it, change to not important
FALSE	FALSE	TRUE	FALSE	Was important, read from it, keep being important
FALSE	FALSE	FALSE	FALSE	Was important, didn't operate, keep being important

For every processor register and for every program instruction except the last one, the robust state of a given register reg in a given point n of a program execution trace

$Trace_{prog \times A}$ will be the result of the robust state of the next point in program execution trace (the previously analyzed instruction) operated with a logical OR with the bits written by the analyzed instruction and then operated with a local AND with the negation of the bits read by the analyzed instruction.

$$1 \leq n < nins(Trace_{prog \times A}); i = f_{prog}(n) \quad (15)$$

$$f_{rstate}(reg, n) = [f_{rstate}(reg, n+1) \vee f_{wbits}(i, reg)] \wedge \sim f_{rbits}(i, reg)$$

When the program finishes its execution we can assume that a change in any of processor registers won't affect the program result anymore. So, we define a function named $f_{endstate}$ that returns a vector with a robust state of a given register with all logical states as true (all register bits classified as unACE).

$$f_{endstate}: ProcReg_A \mapsto \mathbb{B} \quad (16)$$

The f_{rstate} function for each program executed instruction will need the robust state of the next executed program instruction (the previously analyzed program execution trace instruction). In the particular case of the last instruction executed by the program, the f_{rstate} will need the $f_{endstate}$.

$$n = nins(Trace_{prog \times A}); i = f_{prog}(n) \quad (17)$$

$$f_{rstate}(reg, n) = [f_{endstate}(reg) \vee f_{wbits}(i, reg)] \wedge \sim f_{rbits}(i, reg)$$

With all the presented functions in this section it is possible to calculate a program's robustness against transient faults when executed over a determined architecture by calculating the precise amount of unACE bits of the program execution trace. This calculation, by the presented methodology, can be done in a single loop evaluating every program trace instruction backwards.

4 Experimental Evaluation

In order to realize our experimental evaluation we designed a set of experiments to calculate the robustness against transient faults of five programs both using fault injection executions and using the presented methodology.

The selected programs are part of the NAS Parallel Benchmark in its version 3.3. Because of the amount of executions needed to realize this experimental work we choose to evaluate the serial (non parallel) versions of BT, CG, FT, LU and SP benchmarks with their smallest class (S).

All five benchmark programs used in this experimental work were compiled using GNU C and Fortran in their version 4.4.1, with static linkage of libraries used by the programs and with maximum code optimization during compilation (O3).

The computing nodes used in the experiments have Linux Ubuntu Server operating system in version 9.10 with 64 bits kernel in version 2.6.31. The hardware of all computing nodes used have one 2 GHz AMD Athlon 64 X2 processor with 2 gigabytes of memory.

4.1 Results using fault injection

The fault injection environment used in this part of the experimental evaluation uses a tool based on Intel PIN [7] to

flip a single randomly chosen bit of a randomly chosen processor register in an also randomly chosen point of a program execution.

For each one of the evaluated programs we made 8,000 executions with fault injection. The amount of executions was defined with the objective of achieve at least $\pm 2\%$ of confidence interval of the average cases classified as unACE.

In Table II we present the execution time of each benchmark program.

Table II – Benchmark programs execution data.

Benchmark	Execution Time (in Seconds)	Binary Code Instructions	Fault Injection Representation (per Binary Code Instructions)	Instructions Executed	Amount of States to Evaluate	Fault Injection Representation (per Amount of States)
bt	0.19	25.337	31.57%	521.847.689	1.603.116.100.608	0,000004999%
cg	0.16	11.376	70.32%	357.952.094	1.099.628.832.768	0,000007289%
ft	0.28	11.137	71.83%	666.494.276	2.047.470.415.872	0,00000391%
lu	0.08	28.452	28.12%	187.912.097	577.265.961.984	0,000001386%
sp	0.08	21.138	37.85%	212.261.609	652.067.662.848	0,000001227%

Also, we present in TABLE II the amount of instructions executed by each program and the amount of states to evaluate in order to exhaustively cover all possible bit flips in processor registers (amount of instructions executed multiplied by the sum of the amount of bits of all processor registers took into account during the evaluation, in this particular case equal to 3,072).

The Figure 2 presents the robustness calculated with the results of the fault injection executions of the selected programs.

Only one of the evaluated programs, the BT benchmark, didn't achieve 1.5% of standard deviation with 8,000 executions with fault injection.

The CPU time needed to calculate the robustness against transient fault using fault injection executions depends on the fault injection environment used to inject the faults.

The best theoretical amount of time needed can be calculated by multiplying the amount of executions the experiment intends to do (8,000 in our case) by the amount of time needed to execute de program being evaluated once.

Perhaps, the environment we used in our experimentation using fault injections uses dynamic instruction instrumentation during the program execution and adds some overhead to the program execution.

In Table III we present the amount of time we spent to realize all executions with fault injection in our fault injection environment and also how many executions we needed to achieve 2% of standard deviation in robustness.

Table III – Benchmark programs fault injection data.

Benchmark	Execution Time (in Seconds)	Execution Time Using Dynamic Instrumentation (in Seconds)	Fault Injection Time Using Dynamic Instrumentation (in Seconds)	Robustness (Amount of unACE Cases)	Standard Deviation after 8,000 Executions	Executions to 2% of Standard Deviation
bt	0.19	21.73	87.661	55,41%	2,23%	6.346
cg	0.16	11.28	45.749	56,05%	1,31%	3.301
ft	0.28	12.12	49.602	62,34%	1,13%	2.456
lu	0.08	21.31	85.548	39,14%	1,31%	2.019
sp	0.08	16.68	67.043	44,94%	1,68%	3.350

The amount of time needed to realize a program set of executions using our fault injection environment was calculated assuming that the program executes, on average, half of its instructions with the dynamic instrumentation overhead and the other half without any overhead. This is because, once the fault is injected, the environment let the program run until the end without any interference.

4.2 Result using the proposed methodology

In order to calculate the selected benchmarks program's robustness against transient faults using the methodology proposed in this work we used a tool based on Intel PIN [7] to store in a trace file the data collected during a program execution.

In the stored data are the amount and the order of execution of every executed program's basic block, and also all processor instructions that compose all stored basic blocks.

We also developed a program to read the stored program trace and, based on information about how processor instructions deal with registers, calculate the program's robustness of each processor register by analyzing the program trace backwards, as suggests the presented methodology.

In the Table IV we present the time we spent generating the traces and the time to analyze those traces and calculate the robustness with the program that implements the proposed methodology. Also, the table presents the calculated robustness and the total time needed to calculate a program's robustness (time to generate the trace plus time to analyze the trace).

Table IV – Benchmark programs data using our methodology.

Benchmark	Instructions Executed	Trace Generation Time (in Seconds)	Robustness Analysis Time (in Seconds)	Robustness	Total Time (in Seconds)
bt	521.847.367	6.69	413.96	29,88%	420,65
cg	357.952.072	13,27	237,84	57,35%	251,11
ft	666.494.164	6,18	442,65	49,71%	448,83
lu	187.910.412	4,12	139,26	28,93%	143,38
sp	212.261.551	6,01	151,93	39,18%	157,94

The time spent on generating a program trace depends on the program being analyzed algorithm. On the other hand, the time spent on the analysis of the program trace is



Figure 2 – Evaluated benchmarks program's robustness.

proportional to the amount of instructions executed by the analyzed program.

As we already predicted, in Figure 3 we present that the calculated robustness using our methodology is always lower than the calculated using fault injection executions or it can be higher (but almost the same) depending on the amount of executions done to calculate de robustness using fault injection and the random number generator and seed used. Our methodology will score a lower robustness because the approach of using fault injection is more data dependent than our proposal and can mask possible DUE and SDC as unACE as explained previously in section II.

On the analysis of the CPU time spent during the robustness calculation using the proposed methodology in Figure 4, we used on average almost 60% of the time needed to run enough experiments using the best theoretical fault injection method and achieve 2% of standard deviation in the statistical approximation. Also, comparing the CPU time spent during the robustness calculation using the proposed methodology with the real fault injection environment used based on dynamic instrumentation to inject the faults, we needed on average only 1.22% of the time needed to achieve 2% of standard deviation in the fault injection statistical approximation.

All the times collected in our experimental evaluation took in account the use of only one CPU core to all activities without any kind of parallelism. We know that the executions with fault injection are independent and could exploit many cores in processor nodes to minimize the total time needed to calculate a program's robustness against transient faults.

Fortunately, the calculations of each processor register's robustness in the proposed methodology are independent. In this way, we can also take benefit of parallelism to speed up our robustness against transient fault analysis (we could parallelize the analysis of this experimental evaluation in 32 independent threads as we evaluated the robustness of 32 of the processor registers of the experimented architecture).

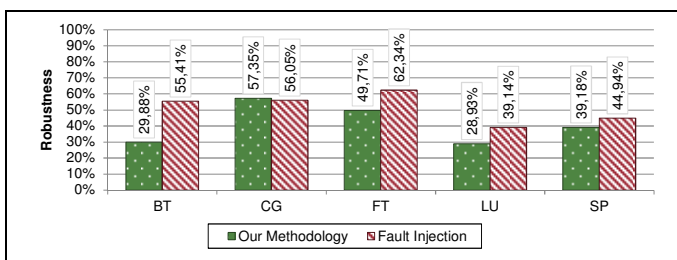


Figure 3 – Our methodology vs. fault injection robustness's.

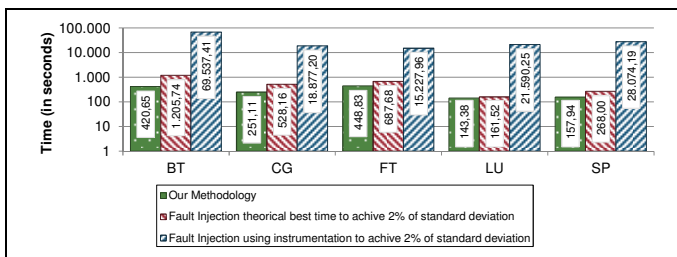


Figure 4 – Time spent on calculating robustness's.

5 Conclusion and Future Work

Evaluate a program's robustness against transient faults by using software based fault injection environments and executing the evaluated program for hundreds or even thousands of times can be a expensive task by the amount of CPU time needed to obtain a statistical approximation of the desired result, even using any type of parallelism.

In this paper we proposed a methodology to calculate a program's robustness against transient faults based on information about the architecture used and on an execution trace of the program running over the architecture.

The proposed methodology calculates the precise amount of unACE bits by analyzing the execution trace. We were able to calculate the robustness almost 41% faster on average than running the programs evaluated with the fastest theoretical fault injection mechanism enough times to score 2% of standard deviation of the unACE cases.

The next step of this work is to improve our methodology program speed by saving time taking into account the repetition of program basic blocks during its execution over a given architecture and exploiting parallelism.

Also, as in this step of our methodology we only classify a program unACE bits, in a next step of our work we will divide the ACE bits in two classifications: DUE and SDC. By knowing precisely the amount of DUE bits of a program will improve even more our robustness evaluation.

6 References

- [1] N. J. Wang, J. Quek, T. M. Rafacz, S. J. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," in Proceedings of the 2004 International Conference on Dependable Systems and Networks, pp. 61—70.
- [2] R. Baumann, "Soft errors in advanced computer systems," in Design & Test of Computers, 2005, vol. 22, pp. 258—266.
- [3] B. Nicolescu, R. Velazco, "Detecting soft errors by a purely software approach: method, tools and experimental results," in Design, Automation and Test in Europe Conference and Exhibition, 2003, pp. 57—62.
- [4] N. Oh, P. Shirvani, E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," in IEEE Transactions on Reliability, 2002, vol. 51, pp. 63—75.
- [5] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, S. S. Mukherjee, "Software-controlled fault tolerance," in ACM Transactions on Architecture and Code Optimization, 2005, vol. 2, pp. 366—396.
- [6] G. A. Reis, J. Chang, D. I. August, R. Cohn, S. S. Mukherjee, "Configurable Transient Fault Detection via Dynamic Binary Translation," in Proceedings of the 2nd Workshop on Architectural Reliability (2006).
- [7] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation" in Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 190—200.

Update and Restructure Legacy Code for (or Before) Parallel Processing

F. G. Tinetti^{1,4}, M. Méndez¹, M. A. Lopez², J. C. Labraga², P. G. Cajaraville³

¹III-LIDI, Fac. de Informática, UNLP, La Plata, Argentina

²Centro Nacional Patagónico, CONICET, Puerto Madryn, Argentina

³Fac. Ingeniería, UNPSJB, Sede Pto. Madryn, Puerto Madryn, Argentina

⁴ Comisión de Investigaciones Científicas de la Prov. de Bs. As., La Plata, Argentina

Abstract—Transformation of an old large sequential applications into a parallel version is still a big challenge in the field of parallel computing. This paper presents a set of transformations to be applied on a medium/large numerical legacy code program having in mind the final objective of parallel computing. The steps are oriented to improve source code while preserving the software external behavior. Each of these transformations has been selected at least to turn old Fortran source code more readable and understandable in order to upgrade it and make it easier to be parallelized.

Keywords: Legacy Code, Parallelization, Fortran Code Upgrade, Numerical Simulation

1. Introduction

Legacy software have has become an issue in many organizations, involving a number of problems and characteristics [26] [8] [6]. The numerical processing field is strongly related to legacy software since

- There is a large number of applications currently in production, being used in a number of organizations and areas such as aerospace, meteorology, etc. [22].
- Mathematical models and computer simulation are being applied since many decades ago.

Furthermore, single CPU (Central Processing Unit) processing (e.g. in terms of Mflop/s or millions of floating point operations per second) is not expected to follow the so-called Moore's Law [21] [12]. There will be almost no improvement in CPU clock rates, instead, it is expected that the number of CPUs or cores will raise at least in the years coming [23] [24]. Multi- and many-core computers are now commonplace, and combinations of multiple multicore processors are now included in medium to large desktop and server computers. This is directly related to legacy code, since it is not longer valid to expect reducing runtime by getting the latest computer [23] [24]. There will be no runtime reduction unless the code is parallelized in some way. Unfortunately, numerical legacy code is amongst the most reactive to change software, and this paper is focused on how to approach these applications in order to be upgraded and parallelized.

Numerical legacy applications are programmed mostly in Fortran for several reasons. Fortran has been and still is one of the most appropriate languages for numerical processing, mostly because numerical processing was about the only one application field by the time Fortran was created [4] [5]. Fortran is one of the first high level programming languages, it is in use and being updated from decades ago [20] [?], unlike most of the current programming languages, including the most popular ones. Fortran has been the first standardized language and, also, it has several standards reflecting its evolution [1] [2] [3] [15] [16] [17].

Legacy software/applications and the environments in which they are used have several features that make it difficult to change/update:

- Either the software documentation is lost, or is outdated, or did not exist at all.
- Today standard software development methodologies and/or tools have not been used for reaching the current version of the software or even the initial version.
- The current version is in fact the result of several not always well documented maintenance/adaptation changes.
- Several developers have been involved, some of them at the initial stages of the development process and others as time and environment progressed. Also, each developer used its own coding style.

Numerical applications have several disadvantages which are combined with the previous ones on legacy code:

- Physical/mathematical models are usually coded in large and hard to read programs, due in part to the combination of the low level programming language abstractions and numerical method/s properties. Most of the numerical problems/properties involved are, in turn, a combination of discrete number representation and numerical method used to compute a solution [18].
- The software developers usually have not been trained in software development tools/processes. Thus, the initial software version contains structures and/or coding specifically oriented towards using a specific computer or computing facility instead of solving a numerical problem. Also, hardware dependent code sections are

undocumented and difficult to identify in the whole application.

This paper proposes a general methodology as well as presents a proof of concept on a specific legacy code, a global climate model (GCM). This program can be used as representative of a medium/large numerical application, since it has

- About About 300 files containing a total of about 58000 lines of FORTRAN 77 source code.
- Approximately 10% of the files are used for defining FORTRAN Common Blocks (global data).
- About 80% of the source code lines containing comments just identify programmer and/or minor code modifications.
- Most of the FORTRAN routines access global data and, also, define aliased data via “equivalence” declarations.

The parallelization process is hard to start in legacy applications, and an incremental process is presented in this paper. Almost every change/update in the legacy code aims to apply parallel processing, either directly or indirectly:

- Enhancing readability allows understanding the code and, thus, makes simpler and less error prone every other code change. Even when enhancing readability does not imply parallel processing *per se*, it is necessary for every further software modification needed for parallel processing.
- Some software sections are almost directly approached for parallel processing, being loops the most clear ones. Loops are the first candidates for parallelization using OpenMP directives [10]. Also, loops are where most calculations are carried out, so they have to be understood in order to transform that processing in shared/distributed memory parallel computers.

2. The General Approach

The general methodology is similar to that in almost every software maintenance process apply a single change plus the necessary software testing with two objectives. Unlike almost every software maintenance, software testing after a change in this context is focused on assuring that the software did not change its behavior. Standard software maintenance tasks are usually focused in the opposite direction: changing the software behavior (for correcting or adding/deleting/changing software functionality) [25] [9] [7]. There is one point in common, though: some standard maintenance tasks are oriented towards enhancing performance, which is almost exclusively the focus of the work in this paper, including software parallelization. There are several distinguishing characteristics of updating and restructuring legacy code for (or previously to) parallel processing:

- There are well known update definitions, specifically in Fortran legacy code. Those updates can be applied even without knowledge of source code, and applying

those changes may produce better knowledge of the software. For example: FORTRAN 77 code is on the so called fixed format, where every line has to begin in a predefined column. In this context, the change from fixed to free format can be applied directly to the whole software, automatically including some code indenting style. It is not necessary to know anything about the software, unlike a *traditional* maintenance task where something about the software should be known in order to be changed/deleted/etc.

- The process can be reduced to those routines or software sections which have the highest processing requirements of the application. It is possible to take advantage of profiling in order to identify routines or code sections with most of the elapsed runtime. However, this will not be used in this paper, since the change process will be applied to all of the application, disregarding those with greater/minor processing requirements. There are mainly two reasons for applying changes to the whole legacy code:
 - Avoiding further mixture of coding styles, since applying changes to only a section or set of routines will result in partially changed software: some sections/routines in old or legacy style and some others with new features/better style.
 - Distributed memory parallel programming usually implies some coding for adding at least the calls to send()-recv() communication routines. Including code in a partially updated/changed legacy software can be worse than doing the same task directly on legacy code without change.
- Some software sections will have much more effort than others. More specifically, loops will be analysed and updated so that they will be easily handled in further parallelization steps.

Specifically, every software change/update will be made in a series of steps:

- 1) Identify and save (e.g. in a software version manager) the current legacy software application/program, which will be taken as the reference. Every change will be accepted/rejected according to its relationship to the reference program.
- 2) Select and apply a specific change/update to be applied to the reference program. A new program version will be produced.
- 3) Check/verify the new program version by comparison with the previous one. Define and apply software testing comparison criterion/criteria in order to accept or reject the new program version. This may/should include a set of test cases if necessary.
- 4) Accept/reject the change according to the previous comparison. An accepted program version will be the candidate as the current version for the next change.

A rejected program version would be:

- Discarded in order to avoid investing more time/effort in a possible useless change.
 - Reviewed in order to find out the problem/s and possible solution/s.
- 5) Document the accepted/rejected change. In case of an accepted change, documentation should include at least a general description of the change plus several (if not all) specific/actual changes. Specific changes are highly prone to be produced automatically (e.g. by a software version manager).

And the complete legacy software update process can be described as an iteration on these steps, each iteration for a different specific change, as shown in Fig 1, which has several points in common with [11] [13] [14]. The first and

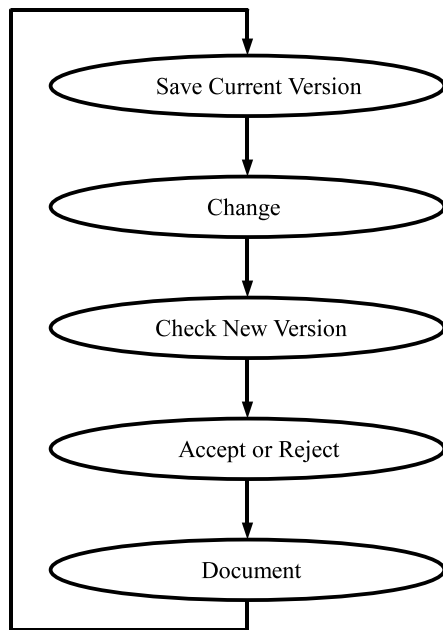


Fig. 1: Software Update Process

last steps can be considered almost automatically made by some well-known tools. The *Change* step on Fortran legacy code has some issues directly related to the language (e.g. old and deprecated Fortran features), others related to the *original* software development tools/methodology (or lack of), and current software version, and others related to the environment for which the updated software is expected to run (i.e. parallel computing hardware). The *Change* step is almost about implementation details of the specific change and the tool selected for implementation. The *Check* step is one of the most interesting steps, entailing software testing. Some software testing is directly related to the application, specifically about analysing and comparing program output. However, given that some changes are about readability (i.e. mostly syntax) it is also expected *a priori* that the change

on itself can be verified without knowledge of the application/output. The *Accept/Reject* involves a difficult decision in numerical applications: whether something changes in the output it is *safe* to accept the new version or not. This is particularly difficult because numerical code is frequently used for simulating a system, and numerical models do not necessarily have a unique (*correct*) output. Specific decisions and implementations of these steps on the CGM described above are shown in the next section.

3. Defining and Verifying Changes

There are several issues/decisions involved in the legacy software update process as described in the previous section. At the highest level of abstraction, some decisions are related to the software feature which needs improvement. At the lowest level of abstraction, there are also several choices such as the tool/s used in the process. This section is focused in the most time and effort consuming updating process steps: *Change*, *Check*, and *Accept/Reject*.

3.1 Possible Changes: Update/Restructure

The first changes applied to the legacy CGM described above have a dual purpose: enhance readability and highlight issues relevant to the parallelization process. Many of the parallelization issues are related to data involved in each calculation, whether it is local to a routine or used from global memory (Fortran Common blocks). Beyond the classical side effect problems, global memory usually makes harder the parallelization process. More global memory accessed in more routines necessarily implies more data traffic (communications) in distributed memory parallel computers such as clusters. Furthermore, more data traffic in general also implies performance penalizations since the multi-core caches become insufficient and/or dirty more often at runtime. Also, some changes are related to old and/or bad coding practices, sometimes accepted by Fortran compilers as *language extensions*. Surprisingly, most of those bad coding practices are not accepted in almost any of the standards, but just as *language extensions* and mostly non-portable features.

- 1) **Remove Tabs:** The tab character is not a legal character in any Fortran standard source code. The GCM as well as other legacy code include such characters and are accepted by compilers via the so called Fortran compiler or *language extensions*. Also, the tab character is handled rather freely by editors and IDEs (Integrated Development Environments), so it is far from enhancing readability.
- 2) **Change Fixed Form to Free Form:** Fixed form source code format was removed as mandatory from the Fortran standard since Fortran 90 [3]. This old language feature makes difficult the process of reading and understanding source code. Furthermore, fixed format source code is prone to errors since blanks

(white spaces), for example, are not meaningful and can be used (or not used) almost freely in the code. It is worth noting that fixed source form is fully standard compliant [17].

- 3) **Replace Old Style DO Loops:** FORTRAN 66 and FORTRAN 77 do not have an end loop statement, it was introduced in the Fortran 90 standard [1] [2] [3]. As a consequence, DO loops use continue statement as an ending point or use a labeled statement in the worst-case scenario. Furthermore, shared DO loop termination becomes a coding style that hinders program readability. Loops are necessarily analyzed and must be well known, since most of the numerical processing is made in loops. Loops are at the initial focus for using OpenMP, for example, in the context of shared memory parallel computing. In distributed memory parallel computing it is necessary to know every data involved in every loop in order to find out the amount of local and non local data and, thus, the communication needs at each processing step.
- 4) **Replace Obsolete Operators:** Old FORTRAN logical operators (.lt., .eq., etc.) were replaced with those commonly used in modern programming languages: <=, ==, etc. This specific update enhances readability, since those old forms for the logical operators are directly related to low level assembly language (or even machine code).
- 5) **Standardize Input/Output Format:** Formatted I/O using labels with format specifications has been traditionally made in Fortran and is currently found in most of the Fortran legacy code. Labels encourage some level of abstraction and uniform handling for I/O but, also, it is possible to spread format labels across several program sections. When a formatted I/O statement is found in the program, it is necessary to locate the corresponding label defining the specific format. The proposal of *standardizing* I/O format is focused on settling:
 - Every format is defined as a constant (*Parameter*, in Fortran terms) character data (*string*).
 - As a character data, every format is declared in the proper program section, the specification part of a Fortran program unit or subprogram.
 Thus, labels will not be associated to format strings and when reading a formatted I/O line in the program it will be clear where to look for the format definition: the corresponding specification part.
- 6) **Remove Unreferenced Labels:** updating old style DO loops as well as standardizing I/O formats usually leads to avoid labels. In fact, avoiding labels generally enhances legibility, since the code is contained in structured constructs and, thus, easier to follow (with exceptions, of course). Other restructuring tasks such as replacing GOTOs, and remove branch to END IF

statements may also lead to useless labels and, thus, in some way it becomes important to remove unreferenced labels just to *clean up* the code. Also, in a large legacy code it is also possible that some labels had become unreferenced due to previous modifications and/or software maintenance tasks.

Some of these specific updates have already been proposed as so called *Fortran refactorings* [19] and/or directly implemented in tools such as [27].

3.2 Checking a New Version

The new program version generated from a change such as those explained above have several ways for checking, besides the classical verification through a software testing process. The main consideration when updating legacy software is ensuring that behavior does not change at all. However, numerical processing implies carrying out large amount of arithmetic operations on numeric floating point representations, and this also implies a large number of involved errors. There are several *stages* at which a new program version can be verified:

- 1) Equivalent source program: some analysis can be made in order to verify that the new program version has exactly the same semantics as the previous one.
- 2) Identical binary program: for those updates that are directly and exclusively related to syntax, it is *a priori* expected that the binary generated by the compiler is exactly the same.
- 3) Identical program output: this is the most common kind of verification/analysis, and it is specially useful
- 4) Analysis of different program output: many numerical legacy applications include or directly are numerical models (such as the GCM described in this paper), so there is not a unique/exact/correct numerical output. Thus, some different outputs are equally acceptable.

With the exception of the last item above, verification can be made almost automatically or at least aided by tools. The analysis of different program output, though, necessarily implies the task of an expert who decide if the *new* results are acceptable or better/worse than the previous ones. There are, also, differences among those alternative verifications at the implementation level. Analyzing *equivalence* of source programs is not easy even in this case, where changes can be considered minimal. At least, the parse trees should be analyzed and equivalence definition/s should be defined. Instead, the analysis of binary code can be made just by simple tools like the `diff` command. This tool (o similar one/s) can be used also for identical program output checking, at the binary level. The identical analysis (of either program binary or program output) can be characterized as the most strict and automatic verification possible, so it is chosen for the work in this paper.

3.3 Accept or Reject a New Version

Even when the identical analysis has been chosen in this paper, the (subsequent) parallelization process cannot left aside. Parallel processing usually (if not always) involves changing the order in which arithmetic operations are carried out, and this leads to numerical differences due to, for example, rounding. Thus, parallel processing highlights almost immediately floating point representation arithmetic errors/differences and the identical analysis verification becomes almost useless in the update process. There are several alternatives for the correctness verification of a new program version:

- Low level/*internal* checking: identify and check numerical operations (e.g. matrix multiplication and/or method results (e.g. LU factorization) for specific allowed errors (e.g. absolute error less than 10^{-5}).
- Whole output/*external* checking: the numerical error is verified/computed directly on program output.

Both alternatives require an application expert in order to accurately define every accept/reject decision. Numerical errors are objective measures, but the impact on the application is not necessarily directly related to magnitude and/or relative numerical error value/s.

4. Implementing and Applying Changes

There is a major decision to be taken about the way/tool/methodology chosen for change/update/restructure legacy code implementation. As mentioned above, there are some available tools such as [27] and those referred to in [14]. Instead, this work is focused in open source tools and, also, working at the AST (Abstract Syntax Tree) level, for several reasons:

- Open source tools can benefit from previous advance done in the area. Also, open source software is prone to be enhanced/completed by a large community for also a large number of specific updates than those presented in this paper. Restructuring legacy code is a long-term project in general, and the work presented in this paper is just an initial approach.
- Program syntax constructions are better handled at the AST than at the source code level. Program variables, for example, are almost immediately identified at the AST. At the source code level it is necessary to define several *ad hoc* rules in order to differentiate a word inside a comment or belonging to a control structure from a variable.
- Even when a Fortran parser is not necessarily easy to develop, there are some Fortran parsers available (one of them used in this work) and it is not impossible to develop a new one, since Fortran is a well-known language.

Several minor decisions are involved as well, related mostly to specific changes and implementation details.

Photran [28], which is based in the Eclipse CDT (C/C++ Development Tooling) plugin [29], was selected as the so called *refactoring* tool. Furthermore, the current Photran version has already implemented two of the aforementioned update/restructure changes: Remove Tabs and Replace Obsolete Operators. One of the most interesting Photran characteristics is the simple inclusion of new application restructure changes. Photran is under development and has several *minor* failures: fixed form refactoring is unsupported, the Fortran Include lines are not handled in fixed form yet, and so forth. Given that the GCM had several Fortran Include lines, it was decided to replace those lines by the corresponding `#include` compiler preprocessing lines. Thus, the first change applied to the CGM did not have any relationship with a Fortran *deprecated* feature and could not be done in Photran, since Photran is not fully functional for our work (something possible when working with open source software). However, the process described above (as shown in Fig. 1) still applies, it is independent of the tool/s used for making a specific change. Fortran Include lines were replaced using a combination of `sed` (stream editor) and `gfortran` (Fortran compiler) options. Fortran Include lines are rather easily replaced, and the code is strongly changed:

- About 40% of the source code files had Include lines.
- About About 25% of the files became removed/useless.

Also, every change described in section 3.1 was applied via Photran, and two of them were already implemented in that tool. The changes specifically implemented and reported in this paper are Change Fixed Form to Free Form, Replace Old Style DO Loops, Replace Obsolete Operators, Standardize Input/Output Format, and Remove Unreferenced Labels

4.1 The Effect of Restructuring

About 10% of the source code files was changed by the Remove Tab update, and all the source code (100%) was changed from (FORTRAN 77) Fixed Format to (Fortran 90) Free Format. Old Style DO Loops were replaced in about 40% of the source code files, and about 15% of the files were affected by the standardization of I/O format. As a result of the previous two restructuring changes, Remove Unreferenced Labels was applied in about 50% of the source code files. Finally, obsolete operators were replaced in about 50% of the source code files. Fig. 2 shows an example of legacy code in 2-a) and the corresponding code after applying some of the implemented updates in 2-b).

As a part of the research on working with legacy code, some metrics were collected on the GCM shown in Table 1. The first column, **Measure**, shows the selected indexes to be counted on source code, the second column, **Before**, shows the resulting number in the legacy code and the third column, **After** shows the results in the legacy code after the restructuring/updates changes have been applied. Some metrics values show a strong change, such as New Style DO

```

if (cond) then
do 100 l=1,1at
do 100 n=1,2
do 200 k=10,14
do 200 m=1,1on
proc_num(m,k,l,n)=0.0
200 continue
100 continue
endif
a) Legacy

```

```

if (cond) then
do l=1,1at
do n=1,2
do k=10,14
do m=1,1on
proc_num(m,k,l,n)=0.0
end do !updated
end do !updated
end do !updated
endif
b) Restructured

```

Fig. 2: Free Form and *New Style* DO Loop Examples.

loops, from 338 to 2648, Old Style DO loops, from 2338 to 0, and Shared DO loops (termination), from 1049 to 0. Some other legacy code undesirable characteristics remain, such as GOTO, ENTRY, and COMMON. Among those old non-

Table 1: Source Code Metrics

Measure	Before	After
GOTO	257	257
ARITHMETIC IF	56	56
DATA	206	206
ENTRY	4	4
COMMON	1388	1388
LABELS	2173	385
FORMAT	240	0
CONTINUE	1008	132
OBSOLETE Oper.	1202	0
END DO	338	2648
DO	2648	2648
New Style DO loops	338	2648
Old Style DO loops	2338	0
Shared DO loops	1049	0

updated features, the most important ones for parallelization could be approached, such as COMMON, which is strongly related to data distribution necessary in distributed memory parallel platforms (such as clusters).

4.2 Checking

After applying each transformation a verification stage was carried out, as depicted in Fig. 1. Two different types of checks/comparisons were performed:

- An *initial* comparison of programs in binary form. Basically, if the executable program is the same, the results (output data) are going to be the same for the

same input data. Each binary program is compared to the previous version in order to check for changes. Different binary programs do not necessarily lead to different results, so another verification is defined in case of the executable program of the new version differs from the previous one.

- A *final* comparison of program output for a set of input data. No changes are allowed, so output data has to be exactly the same as that for the original (legacy) program.

Table 2 shows the way in which every change applied to the legacy CGM was verified, including the non-planned change of Fortran Include lines replacement. For every comparison, the diff and cmp tools were used (i.e. no *ad hoc* tool/program was necessary). Column **B-Prg** stands for

Table 2: Restructuring Verification

Restructuring	B-Prg	B-Out	A/R
(1) Include lines replacement	F	S	A
(2) Remove Tabs	S	N/A	A
(3) Change Fixed Form to Free Form	F	S	A
(4) Replace <i>Old Style</i> DO Loops	F	S	A
(5) Replace Obsolete Operators	F	S	A
(6) <i>Standardize</i> Input/Output Format	S	N/A	A
(7) Remove Unreferenced Labels	F	S	A

“Binary Program Comparison”, where an F indicates that binary program/executable is different from the previous one, so the test failed, and an S indicates otherwise, the binary produced by the compiler after the source code restructuring is exactly the same as the previous version, so the test succeeded. Column **B-Out** stands for “Binary Data Output” and results were not necessary in case of the previous test succeeded or were successful otherwise. Finally column **A/R** stands for “Accept/Reject” change, and every change was accepted, i.e. the new program has not changed behavior/results.

5. Conclusions and Further Work

For this work, seven restructuring software changes have been applied on a legacy CGM, five of which were proposed and implemented. One of the most significant implemented change is the so called “Change Fixed Form to Free Form”, which has been requested by many Phortran users. It can be considered as a contribution to the Phortran refactoring menu. The complete process has been proposed and successfully applied to every change, even when a large proportion of the source code files have been affected, as described above. The resulting source code has been highly enhanced in terms of readability and control structures (processing) identification, so that further changes are encouraged, specifically those related to parallelization.

Most of the legacy code transformation has been applied using Phortran, which has been proven to be a useful tool in order to handle Fortran source code. However, neither

specific changes nor the entire legacy source code restructuring depends on Photran. It is likely that most of the parallelization process would imply using some different tool, but the legacy code enhancement is considered to be strongly necessary step prior to the parallelization task.

The performance of the GCM has not been changed, the transformations in the source code did not have any effect at all in the program's external behavior. The changes made in the CGM have improved the internal structure by upgrading, making more readable, and more comprehensible the CGM. There are more changes that can be performed in the GCM source code in order to obtain a source code more likely to be parallelized.

References

- [1] American National Standards Institute, X3. 9-1966, American National Standards Institute Incorporated, New York, 1966.
- [2] American National Standards Institute, X3. 9-1978, American National Standards Institute, New York, 1978.
- [3] American National Standards Institute, American National Standard for programming language, FORTRAN - extended: ANSI X3.198-1992: ISO/IEC 1539: 1991, American National Standards Institute, 1992.
- [4] J. Backus, "The IBM 701 Speedcoding System," *Journal of the ACM*, Vol. 1, No. 1, Jan. 1954.
- [5] J. Backus, "The History of Fortran I, II, and III," *ACM SIGPLAN Notices*, Vol. 13, No. 8, Aug. 1978.
- [6] K. Bennett, *Legacy systems: Coping with success*, IEEE Software, IEEE Computer Society Press, Vol. 12, No. 1, 1995.
- [7] K. H. Bennett, V. T. Rajlich, "Software maintenance and evolution: a roadmap", *Proceedings of the conference on The future of Software Engineering*, Limerick, Ireland, June 2000.
- [8] M. L. Brodie, M. Stonebraker, *Migrating legacy systems*, Morgan Kaufmann Publishers, 1995.
- [9] N. Chapin, J. E. Hale, K. M. Khan, J. F. Ramil, W. G. Tan, "Types of software evolution and software maintenance", *Journal of software maintenance and evolution: Research and Practice*, Vol. 13, No. 1, 2001, John Wiley & Sons.
- [10] B. Chapman, G. Jost, R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*, The MIT Press, 2007.
- [11] L. S. Chin, D. J. Worth, C. Greenough, *Code Coverage Analysis for Fortran RAL-TR-2009-019*, Aug. 2009, http://www.softeng.rl.ac.uk/media/uploads/publications/2010/06/RAL_TR_2009_019.pdf
- [12] E. P. DeBenedictis, "Will Moore's Law Be Sufficient?," *SC '04 Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, IEEE Computer Society Washington, DC, USA 2004.
- [13] C. Greenough, *Software transformation from Fortran 77 to Fortran 95 of the SHOCK code*, SEG Note, SEG-N-007-2010, March 2010, <http://www.softeng.rl.ac.uk/media/uploads/publications/2010/03/mmu.pdf>
- [14] C. Greenough, D. Worth. *The Transformation of Legacy Software: Some Tools and a Process*, RAL Technical Report TR-2003 012, 2004.
- [15] ISO, ANSI/ISO/IEC 1539-1:1997: *Information technology - Programming languages - Fortran - Part 1: Base language*, American National Standards Institute, 1997.
- [16] ISO, ANSI/ISO/IEC 1539-1:2004, *Information technology - Programming languages - Fortran Part 1: Base Language*, International Organization for Standardization, 2004.
- [17] ISO, ISO/IEC JTC 1/SC 22/WG 5/N1830, *INTERNATIONAL STANDARD ISO/IEC DIS 1539-1, Information technology - Programming languages - Fortran - Part 1: Base language*, Third edition, 2010.
- [18] E. Loh, "The Ideal HPC Programming Language. Maybe it's Fortran. Or maybe it just doesn't matter." *Queue*, Vol. 8, Issue 6, June 2010.
- [19] M. Méndez, J. Overbey, A. Garrido, F. G. Tinetti, R. Johnson, *A Catalog and Two Possible Classifications of Fortran Refactorings*, Technical Report, Facultad de Informática, UNLP, Department of Computer Science, University of Illinois at Urbana-Champaign, Aug. 2010, <https://www.ideals.illinois.edu/handle/2142/16950>
- [20] M. Metcalf, "The Seven Ages of Fortran", *Journal of Computer Science & Technology (ISSN 1666-6038)*, Vol. 11, No. 1, April 2011, pp. 1-8. Available at <http://journal.info.unlp.edu.ar/journal/journal30/papers.html>
- [21] G. E. Moore, "Cramming more components onto integrated circuits", *Readings in computer architecture*, Mark D. Hill, Norman P. Jouppi, Gurindar S. Sohi (Eds.), Morgan Kaufmann Publishers Inc., 2000, ISBN 1-55860-539-8.
- [22] NASA, *Goddard Institute for Space Studies*, <http://www.giss.nasa.gov/tools/>
- [23] H. Sutter "The free lunch is over: a fundamental turn toward concurrency in software", *Dr. Dobbs' Journal*, Vol. 30, No. 3, 2005, <http://www.gotw.ca/publications/concurrency-ddj.htm>.
- [24] H. Sutter, J. Larus, "Software and the Concurrency Revolution," *Queue*, Vol. 3, No. 7, September 2005.
- [25] E. B. Swanson, "The Dimensions of Maintenance", *Proceeding Second International Conference on Software Engineering*, October 1976.
- [26] S. R. Tilley, D. B. Smith, *Perspectives on Legacy Systems Reengineering (Draft)*, Reengineering Center, Software Engineering Institute, Carnegie Mellon University, 1995.
- [27] <http://www.fortran.com/tools.html>, f2f90.
- [28] Photran - An Integrated Development Environment and Refactoring Tool for Fortran, <http://www.eclipse.org/photran/>
- [29] Eclipse - The Eclipse Foundation open source community website, <http://www.eclipse.org/>

Broadcast and Partial Computing Algorithms for Cholesky Factorization on a Cluster of Multicore Computers

Fernando G. Tinetti¹, Gustavo Wolfmann²

¹III-LIDI, Facultad de Informática, UNLP, La Plata, Argentina

Comisión de Investigaciones Científicas Prov. de Bs. As., La Plata, Argentina

²Lab. de Computación - Universidad Nacional de Córdoba

Av. Velez Sarsfield 1611, Córdoba, Argentina

Abstract—*Data dependences are one of the main linear algebra/numerical methods characteristics preventing parallel computing. Partial Computing is presented in this paper focused on identifying such dependences and looking for alternatives in order to compute in otherwise idle/waiting time. Cholesky factorization is used as a test bed, since it defines what can be referred to as a classical dependency pattern in linear algebra methods. The factorization has a series of steps, some of which enforce the execution in only one processing node and the result should be then propagated to other nodes to be used in the next steps. Identifying and avoiding such sequential computing in a parallel algorithm usually implies enhancing performance. Partial computing aims to make progress in the processing identifying partial results for the future. The data partition and task distribution considered originally can/should be reformulated to allow such partial computing.*

Keywords: Parallel Computing, Cluster Computing, Parallel Performance, Cholesky Factorization, Partial Computing

1. Introduction

Matrix factorization algorithms are used in the resolution of linear equations system because of their reliability as well as well-known performance [7]. There are many factorizations methods, depending on the properties of the system. Among them, some of the most well-known ones are LU, QR, and Cholesky factorization methods. These methods can be considered also as algorithms, since they provide a sequence of well defined computational steps for finding out the corresponding factorization. Also, the tiled implementation of the algorithms are generally made up of a series of iterations, and each iteration can be divided into block factorization and block update tasks [6]. Block factorization usually obtain the block final value, while update tasks compute some parts of the remaining data using the last computed block. Thus, there is a strong dependency among data computed in previous stages (blocks) and data pending for computation.

Overcoming data dependences for matrix factorizations is not a trivial task, since the same final result has to be obtained. Partial computing is proposed in order to identify

and effectively compute partial results, i.e., results or data blocks for which some processing can be made in advance, before every necessary data is available/computed. Thus, the final value is obtained as a sequence of partial computations; some of them defined by the method and some others defined for using otherwise idle/waiting time. In this context, partial computing borrows some ideas from data flow computing [8], but maintaining a direct relationship with matrix factorization methods and high level specification and programming.

From a different point of view, the parallelization process has two main tasks: data distribution and task division as reflected by the so called parallel patterns [12] [13]. Even when factorizations have many tasks to be potentially solved in parallel, data dependency impose waiting and, frequently, there is only one processor effectively computing while all the others are just waiting. As explained above, in tiled implementations of factorizations methods, block factorization has to be completed before block updates, thus, every processor has to wait for the block factorization to be completed and, then, update its data with the resulting factorized block.

Most of the problem lies at data distribution and data availability for partial computing combined with idle/busy processors. Partial computing focuses on taking advantage of idle processing on which at least some of the needed data are available (or can be available via communication/s made in background). This is why partial computing is hard to model in general, and this paper introduces the main ideas behind a successful approach. The Cholesky factorization has been selected due to its processing (dependency) pattern as well as usage on linear algebra applications. There are many parallelization alternatives to be taken into account, which are analysed with test results running on different clusters of multicore computers. Alternative parallelization algorithms are also presented for comparison.

2. Related Work

ScaLAPACK (Scalable Linear Algebra package) is the reference library for distributed memory parallel linear algebra routines [3], as a parallel version of the traditional LAPACK

library [2]. ScaLAPACK is designed using the message-passing programming model, arranging the processors on a 2-D rectangular grid. Data is partitioned into square blocks with a block-cyclic distribution for balanced workload, scalability, and maximum data (re)utilization [5]. Thus, having np processors or computing nodes the number of rows and columns, r and c of the processors grid have to be defined such that $np = r * c$. Also, the ScaLAPACK users have to define the optimum data block size for data reuse in the memory hierarchy of each node as well as for computing/communication ratio. Furthermore, ScaLAPACK is not designed *a priori* to run on an hybrid parallel model, though it is not difficult to implement some explicit adaptations.

The look ahead technique [16] is an option to overcome the strong data dependency by overlapping factorization tasks with update tasks available from previous steps. This technique is based on a non-blocking broadcast message implemented in the multicomputer where it was originally tested. Even though good results were reported for this technique, it is not completely clear how to balance the workload among different tasks. Also, non-blocking broadcast is not available in the most popular MPI (Message Passing Interface) implementations, it is only a suggested feature to be incorporated in future implementations of MPI [9]. More recently, the look ahead technique has been applied on multicore computers [10], where the order of operations in the factorization can be either statically or dynamically modified.

The PLASMA project [11] is specifically focused on multi/many-core architectures on which a pipeline of processing tasks is defined. High performance is expected to be obtained on tiling algorithms, which provide processing fine granularity. The whole processing is presented as a directed acyclic graph of tasks, in which the edges of the graph represent dependencies among them [1]. Tasks and data are assigned to each execution thread in a master/worker style [13] with dynamic scheduling for balanced workload. As data is located in shared memory, there is no problem of data communication. This point of view is very interesting but suitable only for shared memory. Under distributed memory, the master/worker synchronization imposes a high performance penalty for fine grained processing.

A parallel algorithm on distributed memory platforms such as clusters, traditionally uses MPI with either collective (basically broadcast) or point-to-point communications. Message passing algorithms are shown in [16] [17] (and references therein) including discussions on data distribution and balanced workload.

3. Parallel Cholesky Factorization

In this section, the Cholesky factorization and the experimentation hardware are described and, later, the different parallel algorithms are presented along with a discussion on their performance. Parallel algorithms include the one

provided in the ScaLAPACK library [3], a broadcast implementation and two alternatives of partial computing.

3.1 Cholesky Factorization

The Cholesky factorization is a classical problem/method in linear algebra, where a square symmetric positive-defined matrix $A \in \mathbb{R}^{n \times n}$ is factorized as:

$$A = L * L^T \quad (1)$$

where L is a lower triangular matrix, and the elements of matrix L are computed as:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} * l_{jk} \right) / l_{jj} \quad 1 \leq j < i \leq n \quad (2)$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2} \quad 1 \leq i \leq n \quad (3)$$

Clearly, there are specific data dependencies for computing the elements, which *a priori* imply a sequence of computation from the top left to the bottom right *matrix corners*. However, specific calculations may be carried out in several different sequences, given that they fulfil the data dependencies defined in Eq. (2) and Eq. (3) above.

3.2 Experimentation Environment

Three clusters were for running the performance experiments:

- Cluster 1, with six interconnected computers (nodes), two of them are dual quad-core Intel Xeon 5420 processors and 8 GB RAM. The remaining four computers have two dual-core AMD Opteron 2200 processors and 4 GB RAM. Every node has two NIC (Network Interface Cards): Ethernet Gb/s and (Flextronics) Infiniband 4x interfaces, with a nominal bandwidth of 20 Gb/s. Every node have Centos 5.3 operating system. The MPI implementation available is OpenMPI 1.3.4. Compiler used is Sun studio 12.1, including the Sun Sunperf library which, in turn, includes a BLAS and LAPACK implementation.
- Cluster 2, with seven identical nodes. Each node has two quad-core Intel Xeon 5420 processors, 8 GB RAM, Ethernet 100 Mb/s, and Infiniband 20 Gb/s interconnection networks. The installed OS is also Centos 5.3, the MPI implementation is MVAPICH 1.1, and the compiler is Intel ifort 11.0, with Intel MKL, which also includes a BLAS and LAPACK implementation.
- Cluster 3, with 8 identical nodes. Each node has two quad-core Intel Xeon 5420 processors, 16 GB RAM, Ethernet 100 Mb/s, and Infiniband 20 Gb/s interconnection networks. The installed OS is also Centos 5.3, the MPI implementation is OpenMPI 1.4.1, and compiler Intel ifort 11.0, with Intel MKL, (including BLAS and LAPACK).

The so called *hybrid parallelism* (for shared and distributed memory parallel processing) has been set as a requirement for every parallel algorithm [14] [15]. Basically, every parallel algorithm has been implemented as a combination of MPI for distributed memory and threads for shared memory (via the pthreads library or the OpenMP [4] implementation made by the compilers). Local processing is made calling BLAS and/or LAPACK routines such as sgemm(), spotrf(), ssyrk() and strsm() routines which are also used by ScaLAPACK and PLASMA algorithms. Previous work [18] has shown that

- Calling optimized library routines for node local processing provides near optimum speed up.
- The focus must be inter-node parallel processing instead of intra-node parallel processing.

Also, using the same libraries for local processing at each node defines a fair baseline for parallel algorithms comparison.

3.3 ScaLAPACK Performance

The specific ScaLAPACK routine for Cholesky factorization is pspotr() and the corresponding experiment results in Cluster 1 are shown in Table 1. The best results are obtained for data block size $nb = 64$, and the results for $nb = 256$ are also shown in Table 1. Basically, the cluster is used as

Matrix Size	Scalapack nb=64	Scalapack nb=256
12000	13.47	16.69
18000	40.52	64.05
24000	92.40	161.64

Table 1

SCALAPACK RUNTIME (SECS.) IN CLUSTER 1 USING INFINIBAND, 32 MPI PROCESSES.

a homogeneous distributed memory parallel computer with 32 processors.

3.4 Broadcast Based Parallel Algorithm

Parallelizing an algorithm implies defining a data distribution and a task partitioning amongst processors. For numerical problems/methods, the data dependences have to be taken into account. From Eq. (3) it is easy to see that the element in the main diagonal only needs the values previously computed in the corresponding row, then the row block data partitioning seems to be a *natural* choice. According to Eq. (2) elements l_{ij} below the main diagonal require elements from two rows, i and j (not the complete rows, though). Thus, if the row block data partition is maintained, row j (or the row block containing row j) should be transmitted to the process containing row i .

In terms of sequential computing steps, computing the elements of matrix L from the initial elements of matrix A can be roughly described as in Table 2, which also provides

Step	Processing
1	Compute the value in the main diagonal l_{11}
2	Use l_{11} for computing $l_{j1}, 1 > j \geq n$
3	Compute the value in the main diagonal l_{22}
4	Use l_{22} for computing $l_{j2}, 2 > j \geq n$
5	Compute the value in the main diagonal l_{33}
6	Use l_{33} for computing $l_{j3}, 3 > j \geq n$
...	...

Table 2

SEQUENTIAL COMPUTING STEPS

a *natural* basis for a broadcast based parallel algorithm. Also, note that l_{ij} can be used as single elements as well as square submatrices/blocks. Clearly, elements in the main diagonal are used to compute the elements in the same column and in subsequent rows: when row i is computed, all the values for column i in the remaining rows can be computed. The algorithm can be specified as an iteration in each process as shown in Fig. 1 with a C-like pseudocode. As an example, Fig. 2 shows the steps of iteration 3, where

```

for i = 1 to n {
  if (row i is local) {
    Compute  $l_{ii}$ 
    Send row  $i$  to other processes
  } else {
    Receive row  $i$ 
    Compute local values at column  $i$ 
  }
}

```

Fig. 1

BROADCAST BASED ALGORITHM: ITERATIONS IN EACH PROCESS.

dark gray indicates computed values before the iteration begins, which are the values at columns 1 and 2. Fig. 2-a) shows the first computing step, in which l_{33} is computed, and Fig. 2-b) shows the second and third steps: broadcast and computing the values at column 3 in every process. Table 3 shows the obtained results in the Cluster 1 with the broadcast algorithm. The best ScaLAPACK results in Cluster 1 are also shown in Table 3 (copied from Table 1) for an easier comparison. It is worth noting that the broadcast algorithm

Matrix Size	Scalapack nb=64	Broadcast
12000	13.47	13.53
18000	40.52	38.08
24000	92.40	82.75

Table 3

RUNTIME (SECS.) OF THE BEST SCALAPACK AND BROADCAST ALGORITHMS IN CLUSTER 1.

was implemented by using the corresponding MPI_Bcast

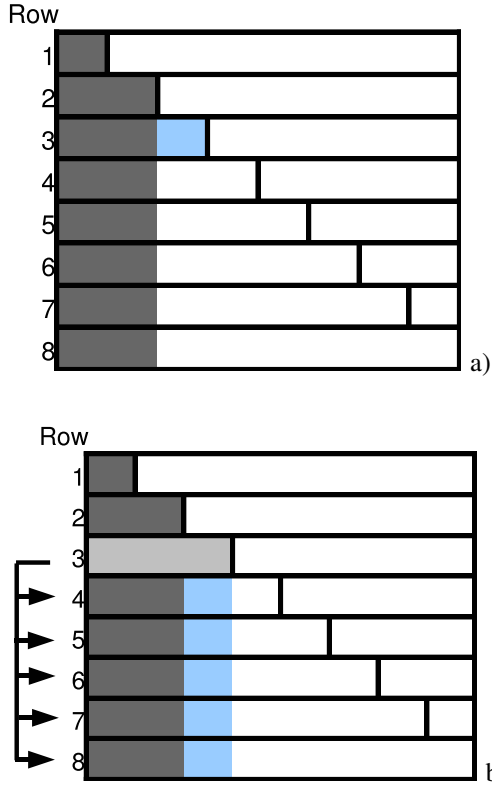


Fig. 2
BROADCAST ALGORITHM AT ITERATION 3.

and the interconnection network used by the MPI library is the same. Performance results for the Broadcast based algorithm are slightly better than those obtained by ScaLAPACK. ScaLAPACK and broadcast algorithms have similar performance, but it does not mean that this is an *acceptable* or *good* performance. Tracing and profiling tools become highly useful in order to analyze and optimize parallel algorithms, since those tools allow to identify “hot spots”, bottlenecks, synchronization penalties, etc. Sun Performance Analyzer is a free tool included in the Sun compiler suite, which allows to obtain a graphical runtime profile and it was used for this specific algorithm. Fig. 3 shows the runtime behavior of the broadcast based algorithm in Cluster 1 for a matrix of size $n = 18000$: vertical lines represent synchronization points/broadcast messages, light gray areas represent computing and light blue (darker gray in black and white printing) represent the time during which processors are idle. Clearly, there is a large amount of time with idle processors, being the end of iteration $n - 2$ one of the most extreme cases:

- The processor containing the last block must wait for the $n - 1$ iteration to be completed in order to use the corresponding values to compute local columns of related to block $n - 1$.

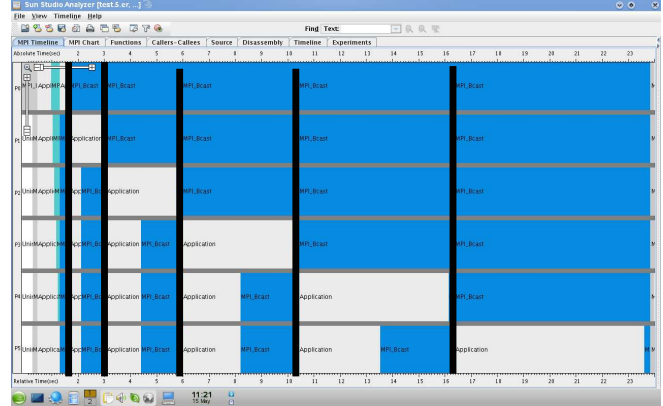


Fig. 3
BROADCAST BASED ALGORITHM PROFILE, $n = 18000$.

- Most of the data available for computing local columns of the last row block related to block $n - 1$ are already computed, but in the process containing row block $n - 1$.
- Most of the imbalanced workload of the last iteration are due to computation which can be made in previous iterations.

3.5 Partial Computing Underlying Ideas

Partial computing follows the terminology of PLASMA, which is focused on identifying computing tasks. In this context, a computing task is just a unit of work, and may involve one or many scalar or submatrix operations. Partial computing divide the whole processing in terms of tasks that compute a partial result and hold these results until the final value is computed when all the needed data are available. The tasks are defined so that partial results become available for other partial results in an otherwise idle processor. For a matrix with $n = 8$, and taking into account Eq. (2), $l_{8,7}$ is obtained as

$$l_{8,7} = (a_{8,7} - l_{8,1} * l_{7,1} - l_{8,2} * l_{7,2} - \dots - l_{8,5} * l_{7,5} - l_{8,6} * l_{7,6}) / l_{7,7}$$

At the end of the second iteration, for example, $l_{8,1}$, $l_{7,1}$, $l_{8,2}$, and $l_{7,2}$ are completely computed, thus the partial involving those elements can be carried out. In general, at the end of iteration h , $1 \leq h \leq j - 1$, all the values of the columns up to h are computed, and the final value for l_{ij} can be unfolded as:

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^h l_{ik} * l_{jk} - \sum_{k=h+1}^{j-1} l_{ik} * l_{jk} \right) / l_{jj} \quad (4)$$

$$1 \leq h \leq j - 1$$

where the first subtraction can be computed, which involves values in columns 1 to h . Thus, every idle processor is able to compute a partial result on local l_{ij} provided it receives

the corresponding values of row j . Elements in the main diagonal are computed according to Eq. (3), and partial computing can also be applied for those calculations. Data dependences are simpler for elements at the main diagonal, since the required data elements belong to the same row, so no communication is involved for a row block data distribution. Partial computing can be identified as in the previous case, for elements below the main diagonal, by identifying idle processors at each iteration. At the end of iteration h , processors with elements at rows $h + 2, \dots, n$ could make in advance some partial calculations for their elements at the diagonal but, instead, they wait for the whole elements of the corresponding rows to be fully computed.

Partial computing on the main diagonal values will be implemented and experimented with in a first stage, given that its implementation is straightforward, i.e. the implementation does not involve any communication. Later, partial computing will be applied in order to compute other element/s of matrix L .

3.6 Partial Computing on the Main Diagonal

At the end of iteration h it is possible to use some of the computed values in row i so that the additions in Eq. (3) can be unfolded as

$$\sum_{k=1}^{i-1} l_{ik}^2 = \sum_{k=1}^h l_{ik}^2 + \sum_{k=h+1}^{i-1} l_{ik}^2 \quad (5)$$

where the elements l_{ik} , $1 \leq k \leq h$ are already computed and the elements l_{ik} , $h + 1 \leq k \leq n$ are not yet computed. Thus, partial computing with elements l_{ik} , $1 \leq k \leq h$ can be immediately made on each processor without any additional data communication.

Partial computing on the elements in the main diagonal is included in the broadcast based parallel algorithm almost straightforward. Table 4 includes the results obtained in Cluster 2 for different combinations of algorithms and

Matrix Size	Bcast-E	PCompD-E	Bcast-I	PCompD-I
12000	10.99	9.41	7.29	5.82
18000	30.53	25.73	22.42	17.56
24000	62.95	52.03	48.48	37.50

Table 4

ALGORITHMS RUNTIMES (SECS.) USING ETHERNET AND INFINIBAND IN CLUSTER 2.

available interconnection networks:

- Bcast-E: broadcast based algorithm using Ethernet Gb/s interconnection network.
- PCompD-E: broadcast based algorithm with diagonal values partial computing using Ethernet Gb/s interconnection network.
- Bcast-I: broadcast based algorithm using Ethernet Infiniband interconnection network.

- PCompD-I: broadcast based algorithm with diagonal values partial computing using Infiniband interconnection network.

Table 5 shows the runtimes for the algorithms in Cluster 1 and Cluster 3 using Infiniband for every data communication, where

- Bcast-i is the broadcast based algorithm in Cluster 1 and Cluster 3.
- PCompD-i is the broadcast based algorithm with partial computing in the diagonal values in Cluster 1 and Cluster 3.

Matrix Size	Bcast-I	PCompD-I	Bcast-3	PCompD-3
12000	16.95	12.90	4.77	4.34
18000	47.31	37.43	13.19	10.61
24000	105.17	83.86	27.20	21.83

Table 5

ALGORITHMS RUNTIMES (SECS.) USING INFINIBAND IN CLUSTER 1 AND CLUSTER 3.

Results in Table 4 clearly confirm that using Infiniband reduces the runtime, since Infiniband is faster than Gb/s Ethernet. Also, partial computing reduces runtimes in varying amounts: between 14.38% and 17.35% using Ethernet and between 9.01% and 23.89% using Infiniband.

Partial computing on the elements belonging to the main diagonal enhances performance, but there are some more performance issues. as indicated by Fig. 4. More specifically,

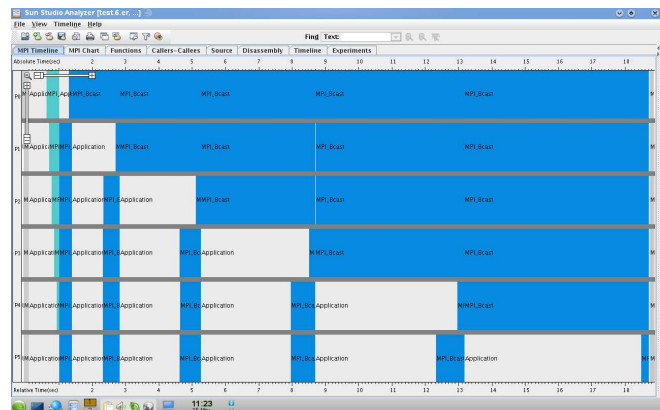


Fig. 4

RUNTIME PROFILE OF THE ALGORITHM WITH PARTIAL COMPUTING IN THE MAIN DIAGONAL, N=18000.

partial computing reduces the last computing period of time (at the bottom right corner in Fig. 4), which is carried out at the node with the last element/s of the main diagonal. However, there is still idle time in which other partial computing can be carried out, as explained in the next section.

3.7 More Partial Computing

The extreme cases of idle and overloaded processors can be easily identified in Fig. 4:

- Processor with the first row block is idle most of the runtime.
- Processor with the last row block is processing most of the runtime.

And this implies a strong unbalanced workload with its corresponding proportional performance penalization. Partial computing is proposed for solving this unbalanced workload.

Most of the work in the processor with the last row block is related to the elements at the main diagonal computing and, the values immediately below the main diagonal. Partial computing is already being done on the elements at the main diagonal, with performance gains. Thus, partial computing on the elements below the main diagonal in the last processor seems a *natural* idea for enhancing performance. Furthermore, the element below the main diagonal requiring most of the arithmetic operations is $l_{n,n-1}$, and taking into account Eq. (4), $l_{n,n-1}$ can be partially computed at the end of iteration h as

$$l_{n,n-1} = \left(a_{n,n-1} - \sum_{k=1}^h l_{nk} * l_{n-1,k} - \sum_{k=h+1}^{n-1} l_{nk} * l_{n-1,k} \right) / l_{n-1,n-1} \quad (6)$$

$1 \leq h \leq n - 1$

and, more specifically, partial computing can be made with all the data available at the end of iteration h , i.e. on

$$\sum_{k=1}^h l_{nk} * l_{n-1,k}$$

Partial computing on $l_{n,n-1}$ should not be made in the node with the last row (it will be called node n), which is already overloaded. Node 1 (the node with row block 1) is the node with less workload (shown at the top of Fig. 4) so it is chosen as the node on which the partial computing will be carried out. Having decided that partial computing will be carried out for element $l_{n,n-1}$ and on node 1, the values involved in the calculations have to be transferred from the proper nodes to node 1. Thus, each iteration will have two more communications among nodes with row blocks $n - 1$ and n (i.e. node $n - 1$ and n respectively) and node 1. At this point, the algorithm with partial computing in diagonal elements and element $l_{n,n-1}$ is completely defined, and the implementation is relatively straightforward.

The algorithm from previous section was modified such that nodes n and $n - 1$ send blocks l_{nh} and $l_{n-1,h}$ to node 1 at the end of each iteration h . Results in Cluster 2 using Ethernet and Infiniband interconnection networks are shown in table 6, where

- PCompD-E: partial computing made only on diagonal values, using Ethernet network for communications.

- PComp2-E: partial computing made on diagonal values and $l_{n,n-1}$ values, using Ethernet network for communications.
- PCompD-I: partial computing made only on diagonal values only, using Infiniband network for communications.
- PComp2-I: partial computing made on diagonal values and $l_{n,n-1}$ values, using Infiniband network for communications.

Note that the algorithm runtimes for partial computing on diagonal values are the same as in Table 4. Table 6

M. Size	PCompD-E	PComp2-E	PCompD-I	PComp2-I
12000	9.41	8.91	5.82	5.21
18000	25.73	23.74	17.56	15.77
24000	52.03	47.84	37.50	33.87

Table 6

PARTIAL COMPUTING ALGORITHMS RUNTIMES (SECS.) USING ETHERNET AND INFINIBAND, IN CLUSTER 2.

again shows that Infiniband network runtimes are better than Ethernet network runtimes, as expected. Also, adding more partial computing also enhances performance between 5.31% and 10,48%. Table 7 show similar results in Cluster 1 and Cluster 3, where

- PCompD-1: partial computing made only on diagonal values only in Cluster 1.
- PComp2-1: partial computing on diagonal values and $l_{n,n-1}$ values in Cluster 1.
- PCompD-3: partial computing made only on diagonal values only in Cluster 3.
- PComp2-3: partial computing on diagonal values and $l_{n,n-1}$ values in Cluster 3.

M. Size	PCompD-1	PComp2-1	PCompD-3	PComp2-3
12000	12.90	11.31	4.34	3.43
18000	37.43	32.79	10.61	8.55
24000	83.86	73.88	21.83	17.65

Table 7

PARTIAL COMPUTING ALGORITHMS RUNTIMES (SECS.) USING INFINIBAND IN CLUSTER 1 AND CLUSTER 3.

Fig. 5 shows the runtime profile for the *new* algorithm (i. e. with partial computing on $l_{n,n-1}$), which is clearly different in several details from that shown in Fig. 4. Node 1 is involved in computing at each iteration and, also, there are more communications, indicated by lines to/from node n and $n - 1$ from/to node 1. There is another interesting detail in Fig. 5: node 1 is assigned more workload, and idle time in node 2 becomes more important from the point of view of unbalanced workload and performance penalty. This may lead to further apply partial computation to other elements of L . The objective is always the same: reduce overloaded

processors calculations by assigning those calculations to idle processors.

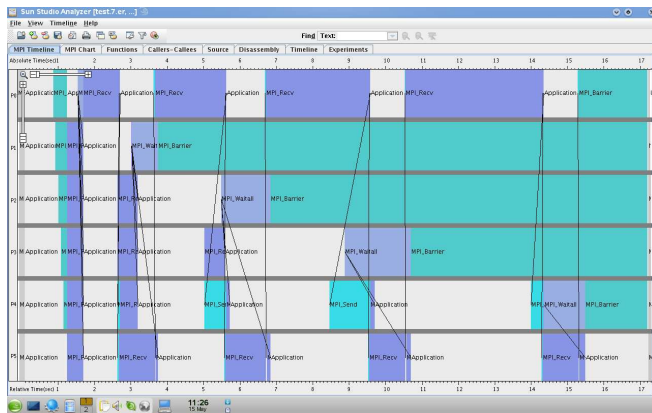


Fig. 5

RUNTIME PROFILE OF THE ALGORITHM WITH PARTIAL COMPUTING IN THE MAIN DIAGONAL AND ELEMENT $l_{n,n-1}$, $N=18000$.

4. Conclusions and Further Work

Partial computing results in performance enhancement for parallel algorithms with specific data dependency. Also, it can be applied iteratively/incrementally according to runtime profile, as shown in the previous section. Basically, available data for partial computing has to be identified, and the necessary communications have to be included in the algorithm in order to *carry out in advance* several operations. Partial computing results can be used on existing parallel algorithms where operations have to be delayed for data availability, as shown for the Cholesky matrix factorization.

Partial computing successfulness depends on the mathematical formulas of the algorithm, data distribution, and configuration of the cluster used for parallel computing. Partial computing may lead to a reformulation of the original data and processing distribution.

The immediate next task in this research involves finding the optimum partial computing usage for the Cholesky matrix factorization from the point of view of performance. Next, other matrix factorizations such as LU and QR should be taken into account. Beyond the algorithm-by-algorithm study, it is expected to define a general model at least for linear algebra operations/methods.

References

- [1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects", *Journal of Physics: Conference Series*, Vol. 180, No. 1, 2009.
- [2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, "LAPACK: A Portable Linear Algebra Library for High-Performance Computers", *Proceedings of Supercomputing '90*, pages 1-10, IEEE Press, 1990.
- [3] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, 1997.
- [4] B. Chapman, G. Jost, R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*, The MIT Press, 2007.
- [5] Jaeyoung Choi, Jack Dongarra, Roldan Pozo, David W. Walker, *LAPACK Working Note 55: ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers*, 1992, University of Tennessee, Knoxville, TN, USA.
- [6] J. Dongarra, D. Walker, "Libraries for Linear Algebra", in *High Performance Computing: Problem Solving with Parallel and Vector Architectures*, G. W. Sabot, Ed., Addison-Wesley Publishing Company, Inc., pp. 93-134, 1995.
- [7] G. H. Golub, C. F. Van Loan, *Matrix Computation*, 2nd ed., The John Hopkins University Press, Baltimore, Maryland, 1989.
- [8] J. Herath, T. Yuba, N. Saito, "Dataflow computing," in *Parallel Algorithms and Architectures*, Lecture Notes in Computer Science, A. Albrecht, H. Jung, K. Mehlhorn, Eds., 1987.
- [9] T. Hoefler, A. Lumsdaine, W. Rehm, "Implementation and performance analysis of non-blocking collective operations for MPI", *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, Reno, Nevada, 2007.
- [10] J. Kurzak, J. Dongarra, "Implementing Linear Algebra Routines on Multi-core Processors with Pipelining and a Look Ahead", *Applied Parallel Computing. State of the Art in Scientific Computing*, 8th International Workshop, PARA 2006, Umeå, Sweden, Revised Selected Papers, 2006.
- [11] J. Kurzak, H. Ltaief, J. Dongarra, R. Badia, "Scheduling Dense Linear Algebra Operations on Multicore Processors," *Concurrency and Computation: Practice and Experience*, Vol. 22, no. 1, pp. 15-44, January, 2010.
- [12] B. L. Massingill, T. G. Mattson, B. A. Sanders, "Reengineering for Parallelism: An Entry Point for PLPP (Pattern Language for Parallel Programming) for Legacy Applications," in *Proceedings of the Twelfth Pattern Languages of Programs Workshop (PLoP 2005)*, 2005. <http://www.cise.ufl.edu/research/ParallelPatterns/plp2005.pdf>
- [13] T. G. Mattson, B. A. Sanders, B. L. Massingill, *Patterns for parallel programming*, Addison-Wesley Professional, ISBN 0321228111, 2004.
- [14] L. A. Smith, *Mixed Mode MPI / OpenMP Programming*, Edimburgh Parallel Computing Centre, Edimburgh, 2000. http://www.cslab.ntua.gr/courses/pps/files/Mixed_Mode_MPI-OpenMP_Programming-Tutorial.pdf
- [15] Lorna Smith, Mark Bull, "Development of mixed mode MPI / OpenMP applications", *Scientific Programming*, Vol. 9, Issue 2, Aug. 2001.
- [16] P. E. Strazdins, "A Comparison of Lookahead and Algorithmic Blocking Techniques for Parallel Matrix Factorization", *International Journal of Parallel and Distributed Systems and Networks*, 4(1), Jun 2001, ACTA Press Calgary, pages 26-35.
- [17] Fernando G. Tinetti, Fernando Romero, "Factorización de Matrices Cholesky: Paralelización y Balance de Carga", *XI Congreso Argentino de Ciencias de la Computación (CACIC)*, Concordia, Entre Ríos, Oct. 2005, <http://tinetti.webs.com/reptec2005-2008/>
- [18] Fernando G. Tinetti, Gustavo Wolfmann, "Parallelization Analysis on Clusters of Multicore Nodes using Shared and Distributed Memory Parallel Computing Models", *Proc. 2009 World Congress on Computer Science and Information Engineering*, IEEE Computer Society, Los Angeles/Anaheim, USA, April 2009, ISBN 978-0-7695-3507-4/08.

Parallel Smith-Waterman Algorithm for DNA sequences Comparison on different cluster architectures

Rucci Enzo, De Giusti Armando E., Chichizola Franco.

Instituto de Investigación en Informática LIDI (III-LIDI) – Facultad de Informática – Universidad Nacional de La Plata. Argentina

{erucci, degiusti, francoch}@lidi.info.unlp.edu.ar

Abstract –DNA sequence alignment is one of the most important operations of bioinformatics. In 1981, Smith and Waterman developed a method for sequences local alignment. However, in practice, various heuristics are used due to the processing and memory requirements of Smith and Waterman's algorithm.

Even though they are faster, heuristics do not ensure that the optimal alignment is found. For this reason, it is interesting to study how to apply the computer power of different parallel platforms to speed up the sequence alignment process without losing result accuracy.

In this paper, a parallelization of Smith-Waterman algorithm is presented using a pipeline scheme due to the data dependencies inherent to the problem. Also, a comparative analysis is carried out regarding the behavior of this algorithm on different multiprocessor architectures: heterogeneous cluster and multicore cluster.

Finally, the results obtained with the different tests performed are presented, as well as future research lines.

Keywords: Bioinformatics, Sequence Alignment, Parallel Algorithms, Clusters, Multicores, Heterogeneity.

1 Introduction

The study of distributed and parallel systems is one of the most active research lines in Computer Science nowadays [1][2]. In particular, the use of multiprocessor architectures configured in clusters, multiclusters, grids and clouds, supported by networks with various characteristics and topologies, has become general, not only for the development of parallel algorithms but also for the execution of processes that require intensive computation and the provision of concurrent Web services [3][4][5][6].

One of the areas of greatest interest and growth in the last few years within the field of parallel processing applications is that of the treatment of large volumes of data such as DNA sequences. The extensive comparison processing required to analyze genetic patterns demands a significant effort in the development of efficient parallel algorithms [7].

Up to some years ago, the idea of a direct application of computer methods in natural sciences was odd and not very convincing. However, it is now evident that any serious advance in our knowledge and understanding of, for instance, the complex mechanisms of the cells, would be impossible without the help of powerful algorithms and fast computers.

DNA is the biological element that differentiates species, or the so-called “types”. Therefore, DNA sequence profiling is carried out as a worldwide effort. With the development of techniques that allow unraveling the information contained in DNA, conditions were favorable for the emergence of bioinformatics, which is a branch that seeks not only to acquire, store and organize the biological information contained in DNA molecules, but also to analyze and interpret these data. It involves the resolution of complex problems using tools provided by computational systems. The diagnosis and treatment of medical conditions, the production of genetically enhanced foods, or the identification of living beings focusing on traceability or paternity systems are, among others, major applications in the area. The more complete the genetic information used for the analysis is, the higher the certainty of the analysis will be.

The center for all bioinformatic operations and analyses is partly held by Sequence Alignment, both for pattern searching among amino acid and nucleotide sequences, and for the search of phylogenetic relationships among organisms. The Smith-Waterman algorithm for local alignment is one of these methods; it focuses on similar regions only in part of the sequences, which means that the purpose of the algorithm is finding small, locally similar regions. This method has been used as the basis for many subsequent algorithms and is oftentimes used as basic pattern to compare different alignment techniques. If the length of the sequences involved are N and M , the complexity of the algorithm is $O(N \times M)$. Thus, the problem is escalated as the square of sequence size [8].

Taking into account that sequences can have up to 10^9 nucleotides each, the time and memory required to solve this problem in a sequential manner is impracticable. This leads to the parallelization of the algorithm over powerful parallel architectures.

1.1 Distributed Parallel Architectures

The term cluster is applied to sets of computers built with standard hardware components that act as if they were an only computer [9]. Nowadays they play an important role for the solution of problems from the science, engineering, and modern commerce fields [10]. Cluster technology has evolved to support activities that go from supercomputing applications and mission-critical software, web servers and e-commerce, to high-performance databases, among other uses.

Clusters computing is the result of the convergence of several current trends, including the availability of cheap high-performance processors and high-speed networks, the development of software tools for high-performance distributed computation, and the growing need for computer power for the applications that require it [11].

Building cluster nodes is relatively easy and economic due to their flexibility: they can all have the same hardware configuration and operating system (homogeneous cluster), or they may have different hardware and/or operating system (heterogeneous cluster). This feature is an important factor when analyzing the performance that a cluster can offer as a parallel machine [12].

The technological change caused by energy consumption and heat generation problems that appear when escalating processor speed has caused the appearance of multicores. This type of processors is formed by the integration of two or more computer cores within the same chip, and increases application performance by dividing computing work among all available cores [13] [14].

The incorporation of this type of processors to conventional clusters gives birth to an architecture that combines shared and distributed memory, known as multi-core cluster [15][16].

1.2 DNA Sequence Comparison on a Multi-core Cluster and a Heterogeneous Cluster

In this paper, the parallelization of Smith-Waterman algorithm is analyzed for the alignment of DNA to determine the similarity degree between two large chains. The behavior of the parallel solution on two distributed architectures is considered: *multi-core clusters* and *heterogeneous cluster*.

In this context, it is important to study algorithm parallelization so that they are efficient in the different cluster-type distributed architectures: partially or totally distributed memory [17][18].

In particular, the approach of the application to study is attractive due to its complexity and the possibility of breaking down parallel algorithm concurrency into "blocks" of different dimensions, which allows an optimal adaptation of the application to the support architecture.

In Section 2, the Smith-Waterman algorithm is explained, together with the sequential and the parallel solutions used in this paper. In Section 3, the experimental work carried out is described, whereas in Section 4, the results obtained are presented and analyzed. Section 5 presents the conclusions and future lines of work in relation to this paper.

2 Smith-Waterman Algorithm Definition

This method allows aligning two DNA sequences by inserting gaps (if necessary) that are used to detect locally similar regions that may indicate the presence of a relation between both sequences, which is done by assigning a similarity score. If gaps are inserted, that is, certain elements of the sequences are not aligned to achieve a better overall alignment, a penalization is applied.

The algorithm calculates a similarity score between two sequences and then, if necessary, employs a backwards alignment process for an optimal result [7].

The following paragraphs explain the operation of the algorithm to find a similarity score between two DNA sequences.

Given two sequences: $A = a_1a_2a_3\dots a_M$ and $B = b_1b_2b_3\dots b_N$, a matrix H of $(N+1) \times (M+1)$ is built, in such a way that the nucleotide bases that form sequence A label the rows (starting with 1), and those from sequence B label the columns (starting with 1). The following steps are applied to calculate the values of H that will yield the similarity score between A and B :

- Start row 0 and column 0 of H with 0, as indicated in Equation 1.

$$H_{i0} = H_{0j} = 0 \quad \text{for } 0 \leq i \leq N \text{ and } 0 \leq j \leq M \quad (1)$$

- Calculate the value of H_{ij} , $\forall i \in [1, \dots, N]$ and $\forall j \in [1, \dots, M]$ by means of Equation 2. This value indicates the maximum similarity between two segments ending in a_i and b_j , respectively.

$$H_{ij} = \max \begin{cases} 0 \\ H_{i-1, j-1} + V(a_i, b_j) \\ C_{ij} \\ F_{ij} \end{cases} \quad (2)$$

- $V(a_i, b_j)$ is the matching function that indicates the score obtained for matching a_i with b_j . It is based on a table of values called *substitution matrix* that describes the probability of a nucleotide base from sequence A at position i to occur in sequence B at position j . The most common matrix is the one that rewards with positive value when a_i and b_j are identical, and punishes with a negative value otherwise.

- C_{ij} is the score in column j considering a gap, and is calculated with Equation 3.

$$C_{ij} = \max_{1 \leq k \leq i} \{H_{i-k,j} - g(k)\} \quad (3)$$

- F_{ij} is the score in row i considering a gap, and is calculated with Equation 4.

$$F_{ij} = \max_{1 \leq l \leq j} \{H_{i,j-l} - g(l)\} \quad (4)$$

- $g(x)$ is the penalization function for a gap of length x , and is obtained with Equation 5, q being the penalization applied for opening a gap and r the penalization for prolonging it.

$$g(x) = q + rx \quad (q \geq 0; r \geq 0) \quad (5)$$

- c. The similarity score is obtained as shown in Equation 6.

$$G = \max_{(0 \leq i \leq N)(0 \leq j \leq M)} \{H_{ij}\} \quad (6)$$

- d. Based on the position in matrix H where the value G was found (representing the end of the highest-scoring alignment between both sequences), a backwards process is performed to obtain the pair of segments with maximum similarity, until a position whose value is 0 is reached, this being the starting point of the segment.

2.1 Sequential Solution of Smith-Waterman Algorithm

In this section, the sequential solution of Smith-Waterman algorithm is analyzed with the purpose of determining the similarity score between two DNA sequences. This means that the backwards process is not taken into account when obtaining the segment that represents the optimal alignment (step d of the algorithm explained in the previous section is not performed).

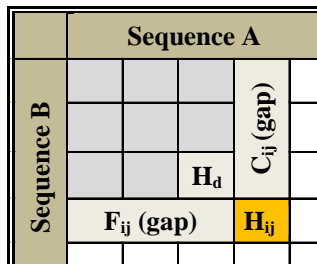


Fig. 1. Data dependency scheme.

Figure 1 shows the data dependency that exists for calculating matrix values. To obtain H_{ij} , the result of $H_{i-1,j-1}$ (H_d in Figure 1) is required, and the score must be known when considering a gap in row i and another one in column j . This restriction allows calculating H values from top to bottom and left to right ($H_{11}, H_{12}, H_{13}, \dots, H_{21}, H_{22}, H_{23}, \dots$).

Taking into account that step d of the algorithm is not carried out, matrix H does not have to be stored in full, all that is needed is:

- A vector h of length $M+1$ that at each position keeps the value obtained in the last processed row over that column. Equation 7 shows the values for h corresponding to the example shown in Figure 1.

$$h_k = \begin{cases} H_{i,k} & k < j-1 \\ H_{i-1,k} & k \geq j-1 \end{cases} \quad (7)$$

- An element e to temporarily store the last value calculated in the row that is being processed. In Figure 1, $e = H_{i,j-1}$.

- A vector c of length $M+1$ that at each position keeps the maximum score considering a gap in that column. Equation 8 shows the values for c corresponding to the example shown in Figure 1.

$$c_k = \begin{cases} C_{ik} & k < j \\ C_{i-1,k} & k \geq j \end{cases} \quad (8)$$

- An element f that keeps the maximum score considering a gap in the row that is being processed. In the example shown in Figure 1, $f = F_{i,j-1}$.

2.2 General Parallel Solution of Smith-Waterman Algorithm

The data dependency mentioned in the previous section causes the problem to be solved following a pipeline scheme where S stages perform the same work over various consecutive nucleotide subsets of the first sequence (A in Figure 1). In each cycle, stage s_i (for $i \in [1, S-1]$) receives a data block from s_{i-1} , solves part of its work, and then sends these results to s_{i+1} (except for the last stage which does not need to send its results to any other stage). The first stage (s_0) only performs its work by sending partial results (corresponding to a block) to its successor.

An important aspect of this solution is selecting the number of elements (BS) from sequence B that form the data blocks that are sent from one process to another, taking into account that:

- Pipeline parallelism is exploited to its maximum capacity only after $S-1$ cycles have been processed. That is, when all stages have received work to do. The larger the BS , the longer the time required to fill the pipe, and therefore, the lower its exploitation. From this point of view, BS should tend to 1.
- If the size of BS is very small, the stages spend more time communicating partial results than actually processing information. From this point of view, BS should tend to N .

A suitable block size should be found, so that data communication and data processing can be done simultaneously. The optimal size does not only depend on the

architecture used, but also on the communication model implemented.

In previous works, a procedure for calculating the optimal value of BS based on architecture characteristics and sequence size has been established [19].

2.2.1 Message Passing as Communication Model

In this case, each pipeline stage is carried out by a different process p_i (for $i \in [0, S-1]$), and partial results are communicated by sending messages between consecutive processes. The first sequence (A in Figure 1) is distributed by p_0 among the S processes that form the pipeline.

2.2.2 DNA Sequence Distribution

For the parallel solution to be efficient, work load should be balanced, that is, all processes should work for the same time.

If the architecture being used is homogeneous, all that is needed is distributing the first sequence (A in Figure 1) equally among the various processes that form the parallel application to achieve a balanced solution.

If an heterogeneous architecture is being used, balancing the work load is more complex because it depends on the power of each processing unit (PU). To do this, two concepts that characterize the architecture being used are applied: the *relative computer power* (rcp) and the *total computer power* (tcp).

Each processing unit i is characterized by its relative computer power (rcp_i) with respect to the most powerful PU , and is calculated as shown in Equation 9.

$$rcp_i = \frac{Power(PU_i)}{Power(PU_{best})} \quad (9)$$

On the other hand, total computer power (tcp) reflects the power of the architecture as a whole with respect to the most powerful processing unit, and is calculated as shown in Equation 10 (assuming an architecture with P processing units).

$$tcp = \sum_{i=0}^{P-1} rcp_i \quad (10)$$

Based on these characteristics, a generic solution can be obtained both for homogeneous and heterogeneous architectures. The distribution of sequences (A and B in Figure 1) is done as follows:

- Process p_0 distributes the elements of the first sequence (A in Figure 1), assigning m_i consecutive nucleotides to p_i , for $i = 0..S-1$. Where m_i is determined based on the relative computer power of

the processing unit where p_i is located (rcp). This is shown in Equation 11.

$$m_i = \frac{M \times rcp_i}{tcp} \quad (11)$$

- Process p_0 sends the entire second sequence (B in Figure 1) to processes p_i , for $i = 1..S-1$.

3 Experimental Work

The language used for implementations is C with the MPI library (*OpenMPI*) to manage communications between processes.

3.1 Architectures Used

To analyze the behavior of the algorithm, tests were carried out on two different architectures:

- A *standard cluster* with three types of monoprocessor machines (26 in total) communicated through a 100 Mbits Ethernet network. The characteristics of each computer type are:
 - *Type 1*: Pentium IV 2.4 Ghz, 1 Gb RAM memory.
 - *Type 2*: Celeron 2 Ghz, 128 Mb RAM memory.
 - *Type 3*: Pentium III 700 Mhz, 256 Mb RAM memory.
- A *multi-core cluster*: Blade with 8 blades, each with 2 quad core Intel Xeon e5405 2.0 GHz processors. Each blade has 2 Gb RAM memory (shared between both processors) and 2 x 6Mb L2 cache for each pair of cores [20][21].

3.2 Tests carried out with the standard cluster

To carry out these tests, first the relative computer power of each type of processor must be determined for this type particular of application. This value is obtained by running the sequential algorithm (with relatively small sequences, $N=M < 2500$) on each type of computer. Table 1 shows the computer power of each type of computer relative to the most powerful PU (type 1).

Type of machine	rcp
Cluster 1	1
Cluster 2	0,8
Cluster 3	0,4

Table 1. Relative computer power of each type of processor.

Tests were carried out on subsets of the 26 computers mentioned in Section 3.1 to obtain the different *total computer powers*: 3.2, 5.8, 12.2, 17, 21. Using 4, 8, 16, 24 and 26 processors, respectively.

DNA sequences of various lengths (20000; 40000; 80000; 320000; 640000) were also taken. Table 2 shows the optimal block size (BS) used in each test in accordance with the function described in previous works [19].

Sequence size	3,2	5,80	12,20	17,00	21,00
20.000	72	63	63	62	64
40.000	72	63	63	62	64
80.000	72	63	63	62	64
320.000	72	63	63	62	64
640.000	72	63	63	62	64

Table 2. Block size BS used in each test run on the heterogeneous cluster.

3.3 Tests carried out with the multicore cluster

Tests were carried out using different numbers of processing units (cores): 4, 8, 16, 32 and 64. Sequence sizes were the same as those used for the tests mentioned in 3.2 [22].

All cluster cores being equal, the total computer power of the architecture is given by the number of cores used, since $rcp_i = 1$ for all processing units.

Table 3 shows the optimal block size (BS) used in each test in accordance with the function described in previous works [19].

Sequence size	4,00	8,00	16,00	32,00	64,00
20.000	8	8	40	41	44
40.000	8	8	40	41	44
80.000	8	8	40	41	44
320.000	8	8	40	41	44
640.000	8	8	40	41	44

Table 3. Block size BS used in each test run on the multicore cluster.

4 Results

To assess the behavior of the algorithms developed when escalating the problem and/or the architecture, the speedup and *efficiency* of the tests carried out are analyzed [1][3][11][23].

The speedup metrics is used to analyze the algorithm performance in the parallel architecture as indicated in Equation (12).

$$Speedup = \frac{SequentialTime}{ParallelTime} \quad (12)$$

In heterogeneous architectures, the “*Sequential Time*” is given by the time of the best sequential algorithm executed in the machine with the greatest calculation power. “*Parallel Time*” is the end time for the entire system.

To assess how good the speedup obtained is, efficiency is calculated. To this aim, the speedup obtained is compared with the total computing power (tcp) of the architecture upon

which work is being carried out (which determines the theoretical speedup), as indicated in Equation (13).

$$Efficiency = \frac{Speedup}{tcp} \quad (13)$$

Table 4 shows the efficiency achieved by the algorithm on the heterogeneous cluster for the different sequence sizes and processor subsets. Figure 2 is a chart representation of those same values.

From the results shown in the chart, it can be seen that the algorithm obtains a good efficiency, especially considering the interaction model used (pipeline). This efficiency remains within the range [0.9...0.97], except when using small sequence sizes in large architectures. This is because the ratio between the time during which the pipeline is not working in all its stages and the time the pipeline is full is greater, since there are many stages and less data to be processed.

As it is to be expected, when problem size increases, efficiency also increases, whereas when the total computer power of the architecture (tcp) increases due to a larger number of processors being used, efficiency decreases.

Sequence size	3,20	5,80	12,20	17,00	21,00
20.000	0,95	0,94	0,83	0,77	0,83
40.000	0,97	0,96	0,93	0,90	0,91
80.000	0,96	0,96	0,95	0,94	0,92
320.000	0,95	0,96	0,95	0,95	0,93
640.000	0,95	0,95	0,95	0,94	0,95

Table 4. Efficiency achieved during the tests run on the heterogeneous cluster.

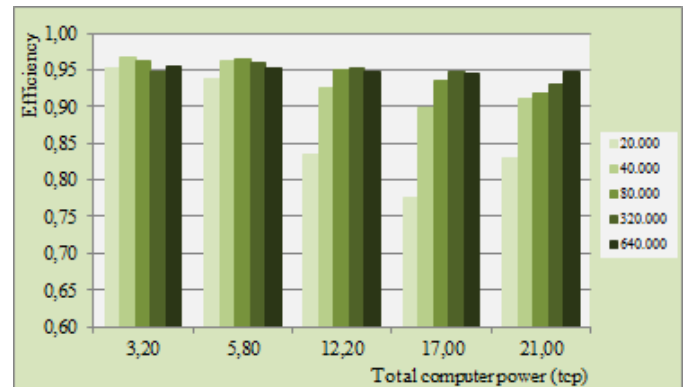


Fig. 2. Efficiency achieved during the tests run on the heterogeneous cluster.

Table 5 shows the efficiency achieved by the algorithm on the multicore cluster for the different sequence sizes and core subsets. Figure 3 is a chart representation of those same values.

This chart shows that the multicore cluster offers a greater increase in efficiency when sequence size increases, and similarly, there is a greater decrease in efficiency when the number of cores in the architecture increases.

If we compare the behavior of the algorithm in both architectures, the multicore cluster offers a better performance than the heterogeneous cluster. This is partly due to the fact that communication speed is higher in the multi-core cluster, which allows working with smaller block sizes BS . By reducing block size BS , the pipeline is full during more cycles, which results in a better exploitation of the parallelism.

Sequence size	4,00	8,00	16,00	32,00	64,00
20.000	0,97	0,97	0,94	0,85	0,76
40.000	0,98	0,97	0,96	0,91	0,83
80.000	0,96	0,98	0,96	0,94	0,92
320.000	0,98	0,98	0,96	0,95	0,92
640.000	0,98	0,98	0,97	0,96	0,95

Table 5. Efficiency achieved during the tests run on the multicore cluster.

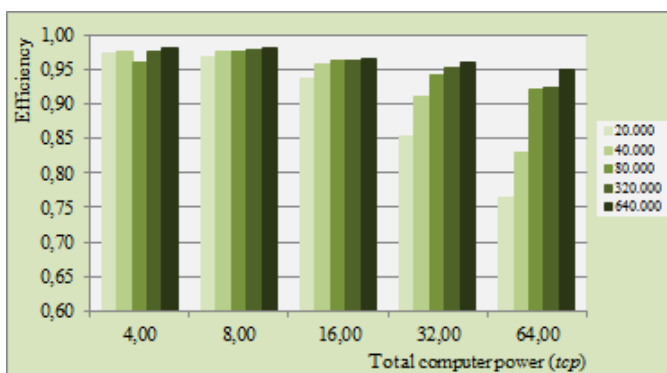


Fig. 3. Efficiency achieved during the tests run on the multicore cluster.

5 Conclusions

In this paper, Smith-Waterman algorithm is parallelized for the alignment of DNA sequences by means of a pipeline scheme due to the dependency of data that is inherent to the problem. A multicore cluster (8 blades with 8 cores each) and a heterogeneous cluster of monoprocessor computers (26 in total, 3 different types) are used as experimental architectures.

The behavior of this algorithm running on these architectures was analyzed, and it was observed that both cluster types yield high efficiency levels. The multicore cluster is slightly superior because its faster communication speed between processes reduces the overhead.

On the other hand, the lack of load balance caused by the lack of accuracy when calculating the relative computer power of each type of processor of the heterogeneous cluster causes the yield obtained with this type of architecture to slightly drop.

Future lines of work focus on three aspects:

- Solution scalability studies (in relation to sequence size and number of processors in the cluster).

- Analysis of the impact of heterogeneity when escalating the problem.

6 References

- [1] Grama A., Gupta A., Karypis G., Kumar V., "An Introduction to Parallel Computing. Design and Analysis of Algorithms. 2nd Edition". Pearson Addison Wesley. 2003.
- [2] Ben-Ari, M. "Principles of Concurrent and Distributed Programming, 2/E". Addison-Wesley, 2006.
- [3] Dongarra J., Foster I., Fox G., Gropp W., Kennedy K., Torczon L., White A., "The Sourcebook of Parallel Computing". Morgan Kaufman Publishers. Elsevier Science. 2003.
- [4] Juhasz Z. (Editor), Kacsuk P. (Editor), Kranzlmuller D. (Editor), "Distributed and Parallel Systems: Cluster and Grid Computing". Springer; First Edition. 2004.
- [5] Di Stefano M., "Distributed data management for Grid Computing". John Wiley & Sons Inc. 2005.
- [6] Miller M., "Cloud Computing: Web-Based applications that change the way you work and collaborate online". QUE Publishing. 2008.
- [7] Attwood T. K., Parry-Smith D. J., "Introducción a la Bioinformática". Pearson Education S.A. 2002.
- [8] Zhang F., Qiao X., Liu Z., "A Parallel Smith-Waterman Algorithm Based on Divide and Conquer". Proceeding of the Fifth International Conference on Algorithms and Architecture for Parallel Processing. 2002.
- [9] Grid Computing and Distributed Systems (GRIDS) Laboratory - Department of Computer Science and Software Engineering (University of Melbourne), "Cluster and Grid Computing". 2007. <http://www.cs.mu.oz.au/678/>.
- [10] Zoltan J., Kacsuk P., Kranzlmuller D., "Distributed and Parallel Systems: Cluster and Grid Computing". The International Series in Engineering and Computer Science. Springer; 1st edition, 2004.
- [11] Wilkinson B, Allen M, "Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers", 2da Edición, Pearson Prentice Hall, 2005.
- [12] Al-Jaroodi J, Mohamed N, Jiang H, Swanson D. "Modeling parallel applications performance on heterogeneous system". IEEE Computer Society, 2003.

- [13] AMD, "Evolución de la tecnología de múltiple núcleo". 2009. <http://multicore.amd.com/es-ES/AMD-Multi-Core/resources/Technology-Evolution>.
- [14] Burger T. W., "Intel Multi-Core Processors: Quick Reference Guide". http://cachewww.intel.com/cd/00/00/23/19/231912_231912.pdf
- [15] Mc Cool M., "Programming models for scalable multicore programming". 2007. <http://www.hpcwire.com/features/17902939.html>
- [16] Chai L., Gao Q., Panda D. K., "Understanding the impact of multi-core architecture in cluster computing: A case study with Intel Dual-Core System". IEEE International Symposium on Cluster Computing and the Grid 2007 (CCGRID 2007), pp. 471-478. 2007.
- [17] De Giusti L., Chichizola F., Naiouf M., De Giusti A., Luque E., "Automatic Mapping Tasks to Cores - Evaluating AMTHA Algorithm in Multicore Architectures". IJCSI International Journal of Computer Science Issues, Vol. 7, Issue 2, No 1. 2010.
- [18] Bertogna M., Grosclaude E., Naiouf M., De Giusti A., Luque E., "Dynamic on Demand Virtual Clusters in Grids". 3rd Workshop on Virtualization in High-Performance Cluster and Grid Computing (VHPC 08). Spain. 2008.
- [19] Chichizola Franco, "Estudio analítico de TB óptimo en base a características del cluster". Technical report III-LIDI. 2011.
- [20] HP, "HP BladeSystem". <http://h18004.www1.hp.com/products/blades/components/c-class.html>.
- [21] HP, "HP BladeSystem c-Class architecture". <http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00810839/c00810839.pdf>.
- [22] Rucci E., "Comparación de modelos de sincronización en programación paralela sobre cluster de multicores ", Tesina de grado, Universidad Nacional de La Plata, 2011.
- [23] Leopold C., "Parallel and Distributed Computing. A survey of Models, Paradigms, and Approaches". Wiley, New York. 2001.

Parallel Optimal and Suboptimal Heuristic Search on multicore clusters. Performance Analysis

Sanz Victoria, Naiouf Marcelo, De Giusti Armando

III-LIDI, Facultad de Informática, Universidad Nacional de La Plata, La Plata, Buenos Aires, Argentina.

Abstract - Discrete optimization problems are interesting due to their complexity and applications, particularly in robotics.

In this paper, a parallel algorithm that allows finding solutions to these problems, is presented. Then, the modifications that can be applied to it to obtain a second parallel algorithm that finds suboptimal solutions, reducing computation time, are studied.

The algorithms proposed are based on two variations of the heuristic search algorithm Best First Search, and are called A^* and Weighted A^* , respectively. The parallel solutions were implemented using MPI to be run on a multi-core cluster, taking the N^2-1 Puzzle as study case.

The experimental work focuses on analyzing the speedup and efficiency achieved for various initial instances, varying architecture configuration.

Finally, the quality of the solutions found by the optimal and suboptimal algorithms are compared and performance variation is analyzed.

Keywords: Discrete optimization, Heuristic search, Parallel algorithms, Optimal and suboptimal solutions.

1 Introduction

One of the areas of interest in parallel computing in recent years has been search processing in graphs. Discrete optimization problems comprise a large number of areas [1] and are often solved with heuristic search algorithms, variations of *Best First Search*, that browse the graph that represents the state space of the problem starting from an initial state to reach a “solution” state in such a way that a target function is minimized. The heuristic function is used to assess the cost of the states in order to process first those that look more promising [2].

In general, these search techniques are very expensive, both as regards computer time and memory use, because state spaces grow in a factorial or exponential manner. This drives the development of parallel algorithms for optimization problems to achieve efficient solutions [3].

The natural parallelization of the technique on a multiprocessor architecture consists in starting the evolution from different graph nodes on the different processors. Processes need to be communicated to be able to report the partial results achieved as the algorithm progresses so as to enable search termination detection or discard graph branches

that, based on the selected metrics, will not improve the partial solution found so far [4].

Some of the aspects observed when using parallel architectures for the resolution of discrete optimization problems [5] are of interest:

- Parallelization *granularity* (ratio between *independent processing time and communication*) is critical for performance, since it will affect the improvement of solution time as well as communications overhead.
- *Load balancing* has to be dynamic, since the state space is implicit and generated during the search. This requires communication, since exploratory work is variable and very hard to predict *a priori* [6].

Two of the main aspects of performance analysis are the *Speedup factor* (Sp) [7] [8] and the *Efficiency* (E) that relates the Speedup with the number of processors (N) used [9] [10].

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 multiprocessor 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 [11], is of interest.

The maximum theoretical speedup can in some cases be improved, which is known as *superlinearity* (Su). The reasons why Sp can be greater than N , in particular for the resolution of discrete optimization problems, is an issue of interest.

The exploration of the state space can be reduced by distributing the workload between N processes so as to “cut down” or “finish” the global search when reaching the expected result in any of these processes [12] [13]. That is, in theory, the cluster architecture will allow superlinearity depending on workload balancing, processor heterogeneity, and the processing time/communication time ratio of the algorithm used [14].

Even though *single-core* cluster architectures have become common platforms in parallel computing, there is nowadays a growing trend towards using multicores and multi-core clusters.

In general, many of the existing parallel applications that run on a *single-core* cluster use the MPI standard. To adapt to new architectures, the major implementations of the standard, such as Open MPI, have been optimized to take advantage of those communications that can be done through shared memory. [15].

As a consequence, it would be interesting to study the variations in the performance of parallel application that communicate through message passing and are run on multi-core clusters when processes are located on: a) cores of the same chip, so that messages are exchanged through L2 cache; b) different processors in the same node, so that messages are exchanged through shared memory; and c) different cluster computers, so that communication is done through a network.

Depending on the initial state, the search for an optimal solution can require excessive computing time even applying parallelism. In some cases, a suboptimal solution that can be reached faster might be acceptable or even preferable. Heuristic algorithms that search for optimal solutions can be adapted to reduce complexity in time, processing a smaller number of nodes at the risk of reducing the quality of the solution obtained. [16].

2 Contribution

In this paper, a sequential algorithm and a parallel algorithm, based on the A* algorithm, to find optimal solutions to the N^2-1 Puzzle problem are presented. The contributions are:

- Carrying out experimental work based on these algorithms with 4x4 boards and using various multi-core cluster configurations, analyzing the performance obtained in each case.
- Presenting a sequential algorithm and a parallel algorithm, based on the Weighted A* algorithm, that can find suboptimal solutions.
- Analyzing the quality of the solutions obtained by the suboptimal search algorithms, as well as performance variations in contrast with the performance achieved by algorithms that search for optimal solutions for the same instances.

3 Characterization of the N^2-1 Puzzle

The N^2-1 Puzzle problem consists in N^2-1 pieces numbered from 1 to N^2-1 placed on an N^2 -sized board [17]. Each square of the board contains one piece, so there is only one empty square. Figure 1a shows an N^2-1 puzzle with $N=4$.

- A legal movement implies moving the empty square to an adjacent position, either horizontally or vertically, by

moving the piece that was in the newly emptied square to the previous position of the empty square.

- The objective of the puzzle is applying legal movements until the initial board becomes the selected final board (Figure 1.b). The solution to the problem should be the one that minimizes the number of movements required to achieve the final configuration from the initial given configuration.

14	7	8	2
13	11	10	4
9	12	5	
3	6	1	15

a

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

b

Fig. 1. Puzzle 15 boards ($N=4$) a. Initial board. b. Final board.

3.1 Heuristics

Heuristic search algorithms use information about the problem to guide the search process, so they value the nodes based on the application of a heuristic function. Thus, they process first the node that looks more promising. The heuristic value of a node is an estimate and indicates how close it is to the solution node.

A more polished heuristic will carry out estimates that are closer to the real cost; therefore, the algorithms that use it will need to process less nodes [2].

The heuristic used by the algorithms presented for the resolution of the Puzzle problem is a variation of the sum of the Manhattan distance of the pieces with the addition of linear conflict detection among pieces, the detection of the last movements applied, and an analysis of corner pieces. The definition can be found in [18].

4 Sequential algorithm for the search of optimal solutions

A* is a variation of the *Best First Search* technique [19], where each node n is valued in accordance to the cost of reaching it from the root of the search tree ($g(n)$) and a heuristic that estimates the cost to go from n to a solution node ($h(n)$). Thus, the cost function will be $f(n) = g(n) + h(n)$. If the heuristic is admissible (i.e., it never overestimates the real cost), the algorithm A* will always find an optimal solution.

The algorithm keeps a list of unexplored nodes (open list¹) ordered by the value of function f , and a second list of already

¹ The open list was implemented by means of a priority queue whose contents are indexed by a hash table that allows efficient searches of a particular state.

explored nodes (closed list²) used to avoid loops in the search graph. Initially, the open list contains only one element, the initial node, and the closed list is empty.

After each step, the node with the lowest f value (the most promising node) is removed from the open list and examined. If the node is the solution, the algorithm ends. Otherwise, the node is expanded (generating the children nodes by applying legal movements) and added to the closed list. Each successor node is added to the open list if it does not appear on either list, or if it does but its cost value improves that of the previous node.

Since there may be many nodes on the open list that have the same cost, a tie-breaking policy was applied in favor of nodes with a lower h value. It has been proven that A* achieves a better performance using this technique.

Once the node that represents the final state has been found, the sequence of nodes on the optimal path can be obtained by following the sequence of pointers to each parent.

5 Parallel algorithm for the search of optimal solutions

The parallelization strategy consists in keeping local open and closed lists on each process or *worker*. At the beginning, only one of these will work with the initial node, and it will also be in charge of detecting the end of the search. As other nodes are generated, processes will receive them and start working.

All workers search locally, building their own closed list – to avoid locally repeated work – as well as their open list. Also, the values of the solutions found are communicated among them, in order to minimize unnecessary searches.

Since the graph for the problem is implicit and generated during the execution, a dynamic load balancing technique has to be adopted. These strategies are based on the idle processor selecting a work *donor* process. If the latter has work, it sends part of its load to the requesting process. Otherwise, it sends a rejection message, and the idle process looks for another *donor*. The technique used in the algorithm is the *Asynchronous Round Robin* [7]. The quality of the nodes sent has to be considered as well, since, if the nodes sent are known not to lead to a better solution, then the receiver will quickly become idle.

To detect the end of the search in a distributed environment with a dynamic load balancing technique, the modified Dijkstra's Termination Algorithm [20] was used, whose purpose is detecting the state in which processes are idle and there are no messages circulating through the network. For this goal, processes are connected in a ring structure and pass a message called *token* between them.

A global pruning algorithm was used where each of the p worker processes has a value that indicates the cost of the best solution found so far (BSC), which is used to limit the search process. Thus, the nodes to process will be only those whose cost is lower than BSC .

A process that has some work pending on its open list will process at the most a fixed amount of nodes for each iteration (LW), or it will process nodes until it finds a solution or until its open list is empty. Then, the worker receives the costs of the "best solutions" – if there are any – found so far by the other workers and updates its BSC variable as needed.

If the process still has some work pending on its open list, it checks if there are any work requests from other processes, and if there are, it sends the first and last nodes of its open list to the requesting process. It then continues working with its nodes.

If the process does not have any pending work, it will be idle, so it will send a work request to its donor following the ARR algorithm. If the process had found a new solution, it sends the corresponding cost to the other processes. It then waits for the types of messages listed below, which will be processed with no particular order of priority:

- *Work request*: an idle worker selected this process as its donor.
- *Work*: the donor sends the requested work. The process is active again.
- *Rejection of work request*: the selected donor does not have any work. The process must send a work request message to the next donor.
- *Token*: reception of the *token* for termination detection. If necessary, the *token* is updated, and it is passed to the next process. Process 0, upon receiving the *token*, checks if it has to end the search process and, if that is the case, it sends a message to the other processes to inform the end of the computation.
- *New solution found*: if necessary, the BSC variable is updated.

Larger messages are those that transfer graph nodes due to load balancing. The number of nodes to communicate depends on the size of the open list of the donor process and the cost of its nodes, but it cannot be over a fixed maximum.

The solution was implemented in C using MPI for process communication.

² The closed list was implemented by means of a hash table that allows efficient searches of a particular state.

6 Sequential algorithm for the search of suboptimal solutions

The complexity in time of the previous algorithms can be reduced by sacrificing the quality of the solution obtained.

Weighted A* (WA*) is a generalization of A* [21]. It is based on using a cost function $f(n) = w_g g(n) + w_h h(n)$, weighting the value of g and h with a constant weight w_g and w_h , respectively. If $w = w_h/w_g$, an equivalent function $f(n) = g(n) + w h(n)$ is obtained. If w is greater than 1, the search is directed towards the most promising direction, since the nodes that are close to the solution (with a lower value of h) are favored. Even if h is admissible, the cost function becomes inadmissible when adding the weight, so the solutions found may not be optimal. On the other hand, if an admissible heuristic h is used, the cost of the solution found by that algorithm will be smaller or equal to $w * S_{opt}$, S_{opt} being the cost of the optimal solution for the instance. [16]

As the value of w increases, the solution will be found faster (processing a smaller number of nodes) at the expense of worsening the quality of the solution obtained. [22]

In the tests carried out, the tie-breaking policy was not applied to the open list because the weight added to the cost function performs that task.

7 Parallel algorithm for the search of suboptimal solutions

The parallel algorithm proposed for finding suboptimal solutions modifies the parallel algorithm presented in Section 5 so as to include the weight when the cost function is assessed. Additionally, this algorithm ends when the first solution is found (since it is not looking for the optimal solution), so the modified version of Dijkstra's Termination Algorithm is not required.

The cost of the solution found can be better, worse, or equal to that of the solution found by the sequential algorithm WA*, due to the way in which nodes are dynamically distributed among processes, with no great variations observed in the practice.

8 Experimental results

For the tests, a multi-core cluster of 12 machines connected by a 1Gbit Ethernet network was used. Each machine has 2 quad core Xeon 5400-series "Harpertown" E5405 processors and 2GB of RAM. Within the quad core, L2 cache is shared between pairs of cores with a capacity of 6MB; this series in particular does not have L3 cache. The implementation of MPI used is Open MPI.

With the purpose of studying the performance of the optimal solution search parallel algorithm, tests were carried

out with various initial configurations of the 15 Puzzle³, with 4 machines, assigning one process to each machine and varying the LW parameter (100-1000, every 50 units, and 1000-2000, every 500 units). The average speedup achieved was 2.97, and the average efficiency was 0.74. In two of the tests, a superlinear speedup was obtained.

One cause for superlinearity occurs when, during parallel execution, a solution node is reached after examining a lower number of nodes than the sequential algorithm. In BFS algorithms, this anomaly is caused by nodes that have the exact same cost as the solution. Another possible cause is the reduction of data structure size in each process. By distributing work among processors, the depth of each open list is reduced, so insertions and removals are faster.

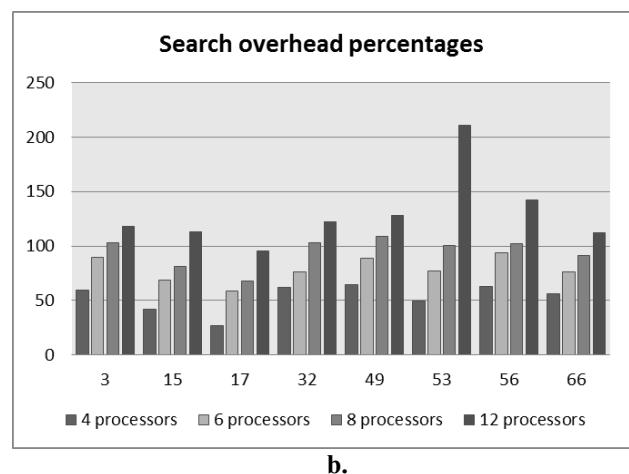
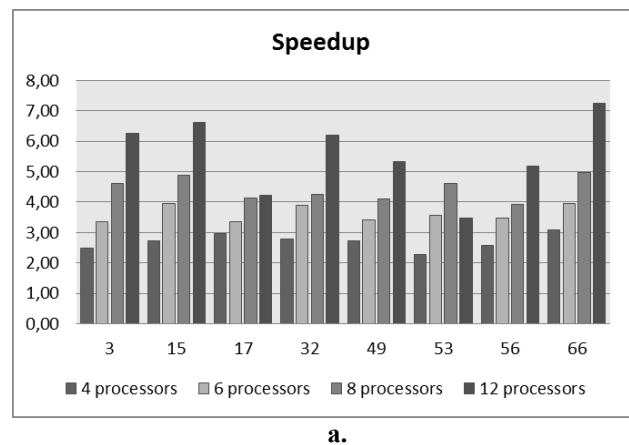


Fig. 2. a) Speedup achieved when escalating the number of processors for various initial configurations b) Search overhead percentages as processors are scaled for the various initial configurations.

³ The initial configurations used are a subset of those presented in article [23]. Only those whose sequential resolution took longer were selected, the final total being 26. The final board used is shown in Figure 1.b.

To observe how this algorithm scales when increasing the number of processors ($P = 4, 6, 8, 12$), it was run assigning one process to each machine and taking into account those instances with the most significant execution times in the test with 4 machines (8 in total). Results are shown in Figure 2.a., where it can be seen that, as P increases for a specific instance, the speedup also increases but the acceleration is not such that it allows keeping a constant efficiency.

The cause of the drop in efficiency was confirmed as the increase in search overhead as the number of processors increases. This factor measures the additional percentage of nodes that the parallel algorithm processes when compared with the sequential algorithm due to speculative expansion (related with the LW parameter). If NP is the number of nodes processed by the parallel algorithm, and NS is the number of nodes processed by the sequential algorithm, the search overhead percentage is calculated as $OB=100*(NP/NS -1)$. Figure 2.b. shows this percentage for the different initial configurations as the number of processors is scaled.

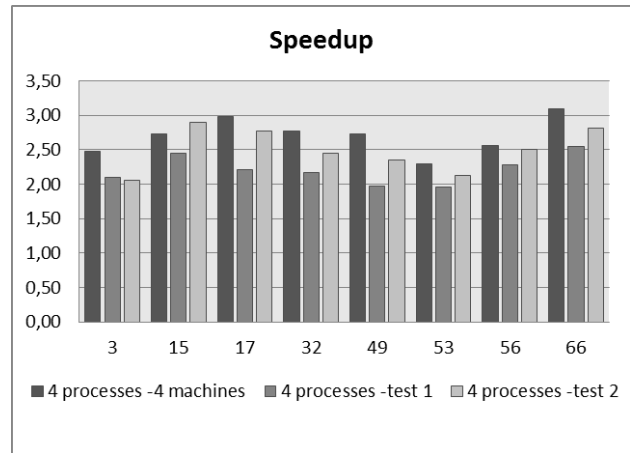
To confirm if the fact that two processes share L2 cache and/or RAM memory would improve performance because messages (in particular, *work* messages) avoid travelling through the network, tests were carried out with 4 processes and 2 machines, with 2 processes assigned to each machine so that:

1. they are in cores that share L2 cache; or
2. they are in two different quad core processors.

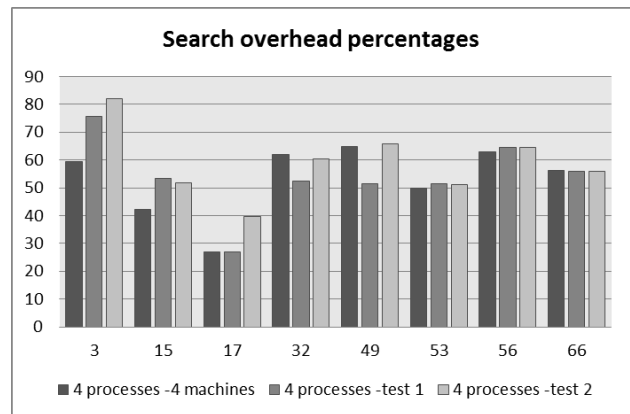
The resulting speedup for both tests is shown in Figure 3 a), and the search overhead is shown in Figure 3 b). As it can be seen, there was no improvement in the performance, with the exception of configuration 15. It should be noted that the size and number of the messages sent by load balance varies because this balance is dynamic. Also, the dynamic load balance can result in finding different optimal solutions for the same instance in two independent runs of the algorithm.

A probable cause for test 1) not to improve its performance may be found in the bottleneck that is produced when accessing the frontside bus of the quad core. On the other hand, both processes mapped to the same machine in test 2) are located in different quad cores and do not compete for the frontside bus, but they do compete for memory controller hub access.

Sequential algorithm WA^* was run with a weight of 1.2, that is, $w_h=6$ and $w_g=5$, for the 8 initial configurations mentioned above and two additional ones for which the execution of the sequential algorithm A^* run out of available memory. An optimal solution was found in all 8 cases. In the two additional cases (configurations 17 and 53), the solution is 6% and 3%, respectively, more expensive than the optimal solution. The reduced computation time in comparison with the execution of A^* is between 81%-97% depending on the initial configuration.



a.



b.

Fig. 3. a) Speedup achieved with 4 processes for test 1 and test 2 b) Search overhead percentage for these tests.

The parallel algorithm that searches for suboptimal solutions was run on 4 machines, with one process assigned to each machine and varying the LW parameter. Considering all testing configurations whose LW reduced times, it was observed that in 8 of them, the cost of the solution found was equal to the cost of the optimal solution.

Initial configuration	Initial disarray	Solution (sequential WA^*)	Solution (parallel WA^*)	Speedup
3	43	59	59	13.98
15	46	62	64 (+2)	12.31
17	46	70 (+4)	70	17.42
32	45	59	59	10.17
49	39	59	59	9.78
53	52	66 (+2)	64	15
56	35	55	55	10.77
60	50	66	66	43.33
66	47	61	61	20.81
88	45	65	65	12.75

Table 1. Solutions found by the sequential algorithm WA^* and the parallel algorithm that searches for suboptimal solutions, together with the speedup obtained.

In particular, an optimal solution was found for instance 53. This may be due to the dynamic distribution of nodes, so that one process found first that solution and caused the search to end. For configuration 15, the parallel algorithm found a solution that is 3% worse than the solution found by the sequential WA*. This information can be seen in Table 1.

The speedup obtained was superlinear in all cases. This is because the parallel algorithm processes a smaller number of nodes than the sequential algorithm.

8 Conclusions and future lines of work

An analysis of the parallel solution to the N^2-1 Puzzle problem that uses message passing as communication mechanism has been presented, and the speedup, efficiency and superlinearity for various multi-core cluster configurations and initial instances have been studied. The modifications to be applied to the previous parallel algorithm to search for suboptimal solutions and thus reduce search time have been presented; optimal or very close to optimal solutions were found in the practice.

Currently, we are studying the migration of the algorithm to use hybrid programming. A possible strategy would be to take into account those *workers* that reside within the same cluster node and have only one shared open list and one shared closed list, implementing these data structures to allow concurrent access.

Finally, scalability studies are being carried out, to increase work volume (N) and the number of processors, on the multi-core cluster to compare the results obtained with those obtained with a cluster of conventional PCs.

9 References

- [1] Sergienko I., Shylo V. Problems of discrete optimization: Challenges and main approaches to solve them. New York: Springer; 2006; 42(4):465-482.
- [2] Russel, S., Norvig, P. Artificial Intelligence, A modern Approach. Pearson Prentice Hall; 2003.
- [3] Ferreira A., Pardalos P. Solving Combinatorial Optimization Problems in Parallel: Methods and Techniques. New York: Springer; 1996.
- [4] Grama A., Kumar V. State of the art in parallel search techniques for discrete optimization problems. IEEE Trans. on Knowledge and Data Engineering, 1999.
- [5] Anderson T., Culler D., Patterson D. A Case for NOW (Networks of Workstations). IEEE Micro 1995; 15(1): pp. 54-64.
- [6] Bohn C., Lamont G. Load Balancing for Heterogeneous Clusters of PCs. Future Generation Computer Systems, 2002; 18(3): 389-400.
- [7] Grama A., Gupta A., Karypis G., Kumar V. An Introduction to Parallel Computing. Design and Analysis of Algorithms. Pearson Addison Wesley; 2003.
- [8] Leopold C. Parallel and distributed computing. A survey of models, paradigms, and approaches. New York: Wiley; 2001.
- [9] Quinn M. J. Parallel Computing: Theory and Practice. McGraw-Hill Companies; 1993.
- [10] Buyya R. High Performance Cluster Computing: Architectures and Systems. Prentice-Hall; 1999.
- [11] Hwang K. Advanced Computer Architecture. Parallelism, Scalability, Programmability. McGraw Hill; 1993.
- [12] Helmbold D., McDowell C. Modeling speedup(n) greater than n . IEEE Trans. on Parallel and Distributed Systems, 1990; 1(2): 250-256.
- [13] Manquinho V., Marques-Silva J. Search Pruning Techniques in SAT-Branch-and-Bound Algorithms for Binate Covering Problem. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, 2002; 21(5): 505-516.
- [14] Sanz V, De Giusti A, Naiouf M. 4-(N^2-1) Puzzle: Parallelization and performance on clusters. Journal of Computer Science and Technology. Vol. 10 Nro 2, pp. 86-90. 2010.
- [15] Open MPI: Open Source High Performance Computing <http://www.open-mpi.org/>
- [16] Ebendt, R., and Drechsler, R. Weighted A* search – unifying view and application. Artificial Intelligence 173:1310–1342. 2009.
- [17] Ratner D. and Warmuth W. Finding a shortest solution for the $N \times N$ extension of the 15-puzzle is intractable. AAAI-86, 168-172
- [18] Korf R. E., Taylor R. Finding Optimal Solutions to the Twenty-Four Puzzle. In: Proc. of the Thirteenth National Conference on Artificial Intelligence (AAAI-96) 1996
- [19] Hart, P., Nilsson, N., and Raphael, B., A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Trans. Syst. Science and Cybernetics, SSC-4(2):100-107, 1968.

[20] Dijkstra E., Scholten C. Termination detection for diffusing computations. *Information Processing Letters* 1980; 11(1):1-4.

[21] Pohl, I. Heuristic search viewed as path finding in a graph, *Artificial Intelligence* 1 (1971) 193–204.

[22] Felner A, Kraus S, Korf R. KBFS: K-Best-First Search. *Annals of Mathematics and Artificial Intelligence* 39: 19–39, 2003.

[23] Korf, R Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, Vol. 27, No. 1, pp. 97-109, 1985.

Parallel Algorithms on Clusters of Multicores: Comparing Message Passing vs Hybrid Programming

Fabiana Leibovich, Laura De Giusti, and Marcelo Naiouf

Instituto de Investigación en Informática LIDI (III-LIDI), Facultad de Informática, Universidad Nacional de La Plata, 50 y 120 2do piso, La Plata, Argentina.
{fleibovich, ldgiusti, mnaiouf}@lidi.info.unlp.edu.ar

Abstract - Given the technological progress of current processors and the appearance of the multi-core cluster architecture, the assessment of different parallel programming techniques that allow exploiting the new memory hierarchy provided by the architecture becomes important.

The purpose of this paper is to carry out a comparative analysis of two parallel programming paradigms- message passing and hybrid programming (where message passing and shared memory are combined).

The testing architecture used for the experimental analysis is a multi-core cluster formed by 16 nodes (blades), each blade with 2 quad core processors (128 cores total). The study case chosen was square matrix multiplication, analyzing scalability by increasing the size of the problem and the number of processing cores used.

Keywords: hybrid programming, cluster, multi-core, message passing, shared memory, parallel architectures.

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 distributed 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].

As previously mentioned, a multi-core cluster is a set of multi-core processors that are interconnected through a network, where they work cooperatively as an only computational resource. That is, it is similar to a traditional cluster but each

node has a processor with several cores instead of a mono-processor.

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.

Memory hierarchy performance is 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). Figure 1 shows a representation of the memory hierarchy in the different architectures.

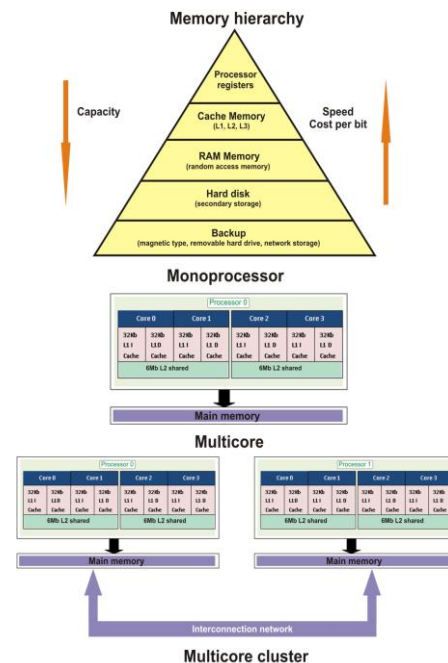


Figure (1). Memory hierarchy

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 core, two memory levels: cache memory shared by pairs of cores (L2) and memory shared among the cores of the multi-core processor [2].

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.

There is a large number of parallel applications in different areas. One of the most traditional and widely studied of these areas in parallel computing, and used in this paper, is matrix multiplication. The reason for using this application (widely tested and assessed) is that it allows using and exploiting data parallelism, as well as analyzing algorithm scalability by increasing matrix size [3]. Thus, the solutions using message passing can be compared with those using shared memory and message passing (hybrid).

This paper is organized as follows: In Section 2, the contribution of this paper is detailed, whereas in Section 3, the features of hybrid programming are described. In Section 4, the study case is detailed, and in Section 5, the solutions implemented and the architecture used to generate the results shown in Section 6 are analyzed. Finally, in Section 7, the conclusions and future lines of work are presented.

2 Contribution

The main contribution of this paper is carrying out a comparative analysis of the performance that can be achieved with hybrid programming in a multi-core cluster architecture versus a traditional parallel programming model (distributed memory).

The analysis is carried out based on running time and efficiency of the hybrid solution as the size of the problem and the number of cores used increase, and the results are compared with solutions that use only message passing.

3 Hybrid Programming

Traditionally, parallel processing has been divided in two large models - shared memory and message passing [1][4].

Shared memory: the data accessed by the application are in a global memory that is accessible to parallel processors. This means that processors can look for and store data from any memory position independently from each other. It is characterized by the need of synchronization in order to preserve the integrity of shared data structures.

Message passing: data are seen as being associated to a specific processor. Thus, message communication among processors is required to access remote data. In this model, sending and receiving primitives are responsible for handling synchronization.

With the appearance of multi-core cluster architectures, a new hybrid programming model comes to existence, which combines both strategies. Communication among processes belonging to the same physical processor can be done by using shared memory (micro level), whereas communication among physical processors (macro level) can be done by message passing.

The purpose of using the hybrid model is exploiting and applying all of the advantages of each strategy, based on the needs of the application. This is a current interest research area; among the libraries used for hybrid programming are Pthreads for shared memory and MPI for message passing.

Pthreads is a library that implements the POSIX (Portable Operating System Interface) 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 [5].

On the other hand, MPI is a message passing interface created to provide portability. It is a library that can be used to develop programs that use message passing (distributed memory) and uses the programming languages C or Fortran. The MPI standard 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 [6].

4 Study Case

Given two square matrixes A and B , matrix multiplication consists in obtaining matrix C , as indicated in equation 1.

$$C = A * B \quad (1)$$

If matrix A has $m * p$ elements, and matrix B has $p * n$ elements, matrix C will have $m * n$ elements.

Each position of matrix C is calculated by applying equation 2.

$$C_{i,j} = \sum_{k=1}^p A_{i,k} * B_{k,j} \quad (2)$$

5 Implemented Solutions and Architecture Used

Experimental tests were carried out based on the implementation of the classical matrix multiplication algorithm, both sequentially and using different parallel programming models: message passing and hybrid (combination of message passing and shared memory).

All three solutions, sequential and parallel, were developed in language C. The parallel solution that uses message passing as process communication mechanism uses the OpenMPI library [6]. The hybrid solution uses the Pthreads library [5] for shared memory and OpenMPI for message passing.

This initial phase of the investigation consists in carrying out an experimental analysis of the behavior of a hybrid application in a multi-core cluster architecture from the point of view of programming models [7][8][9].

The results shown are focused in analyzing the hybrid solution in two aspects:

1. Analyzing behavior when the size of the problem and the number of cores increase (scalability) [7][8]. In this case, square matrixes of 1024, 2048, 4096 and 8192 rows and columns were processed.
2. Comparing running times and efficiency with those obtained with the message passing solution.

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 [10][11].

In the following paragraphs, the solutions implemented are described. In all cases, matrix multiplication is carried out by storing matrix A by rows and matrix B by columns in order to use local cache memory for data access and take advantage of the architecture on which algorithms were run.

5.1 Sequential Solution

Each position of C is calculated as established in equation 1.

5.2 Message Passing Solution

In this case, processing is divided in blocks of rows, which are assigned equally to each process. If p is the number of processes and $n * n$ is the dimension of matrixes A and B , the number of rows of matrix C calculated by each process is n/p .

The algorithm uses a hierarchical master/worker structure. There is a general master that divides all rows that will be processed in each blade, and sends the corresponding rows to the master in each blade. It then behaves as the second level of workers described below. Finally, it receives the results obtained by all application workers.

On the other hand, there is one master in each blade (second-level masters), responsible for receiving the rows that will be solved by the processes in its blade and distributing them among its workers to then process its own share, also acting as a worker.

It should be noted that each process must store the rows from matrix A to be processed, all of matrix B and the rows from matrix C that it generates as a result.

5.3 Hybrid Solution

In this solution, there is one process per blade that internally uses 7 threads to carry out processing activities, and 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 process master.

The algorithm can be summarized as follows:

Master process:

1. *It divides the matrix into blocks of n rows/number of blades used for processing*
2. *It communicates the corresponding rows from matrix A and all of matrix B to worker processes.*
3. *It generates the threads and processes its own block*
4. *It receives results from worker processes.*

Worker processes

1. *They receive the corresponding rows from matrix A and all of matrix B .*
2. *They generate the threads to process the data.*
3. *They communicate the results to the master process.*

6 Results obtained

In the following paragraphs, the results obtained in the experimental tests carried out are presented.

Table 1 shows running times for the sequential solution (Seq.), the message passing solution using 16, 32, and 64 cores (MP 16, MP 32 and MP 64) and the hybrid solution with 16, 32, and 64 cores (H16, H32 and H64). For the tests, both the dimension of the matrix and the number of cores are escalated. Figure 2 shows the speedups obtained for these tests.

It can be seen that the running times obtained by the hybrid solution are always lower than those obtained by the message passing solution. Also, as problem size increases, the time difference between both solutions also increases in favor of the hybrid solution.

Size	Seq.	MP 16	H 16	MP 32	H 32	MP 64	H 64
1024	12.47	0.88	0.86	0.52	0.50	0.73	0.39
2048	101.21	6.72	6.63	3.66	3.58	2.51	2.36
4096	808.57	52.54	51.89	27.42	26.91	17.51	16.06
8192	6479.43	1059.52	410.36	638.87	209.36	752.07	124.89

Table (1)

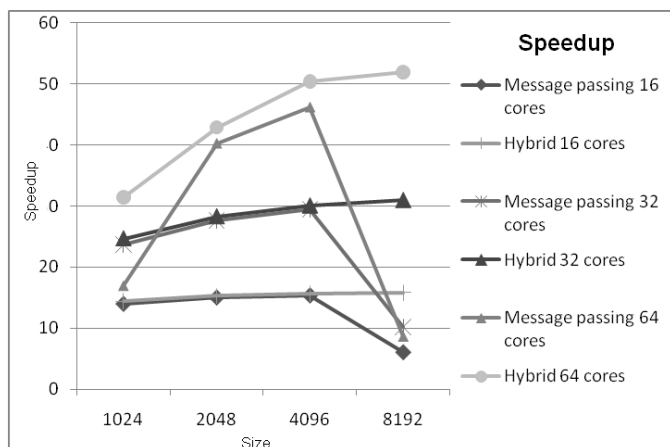


Figure (2). Speedup

6.1 Comparison of Results

Table 2 shows the efficiency achieved by the different testing alternatives; whereas in Figure 3, a comparative chart showing that information is presented.

Based on the results obtained, two observations can be made - the efficiency achieved by the hybrid solution is in all cases higher than the one achieved by the message passing solution, and, as problem size increases (for the same number of processing units), efficiency also increases. However, as it is to be expected, when the number of processing units increases, efficiency decreases due to the increased volume of communications and synchronization among processes.

It should also be mentioned that the efficiency achieved by the message passing solution for 8192 * 8192 elements is significantly degraded in comparison with the other sizes. This is due to limitations in the main memory that is available in each blade, which, for large sizes, generates a swapping of the necessary data structures.

Size	MP 16	H 16	MP 32	H 32	MP 64	H 64
1024	0.87	0.90	0.73	0.77	0.26	0.49
2048	0.94	0.95	0.86	0.88	0.62	0.66
4096	0.96	0.97	0.92	0.93	0.72	0.78
8192	0.38	0.98	0.31	0.96	0.13	0.81

Table (2)

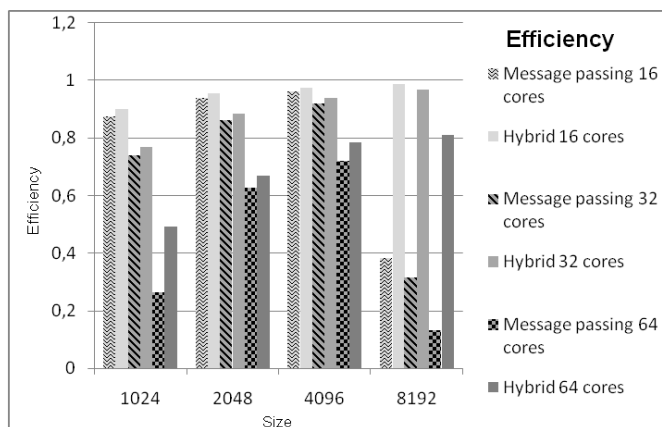


Figure (3). Efficiency

7 Conclusions and future work

As regards scalability, the results obtained show that the hybrid solution is scalable and that an increase in problem size also increases the efficiency achieved by the algorithm.

On the other hand, when comparing the message passing solution versus the hybrid solution, it can be seen that the latter offers better running times.

In this regard, there is improvement introduced by the hybrid solution, which takes advantage of the characteristics of the problem and the architecture used. The possibility of using shared memory makes it unnecessary to replicate data in each blade. In the case of the problem that was chosen as study case, matrix B does not have to be replicated in each of the workers. This does not happen with the message passing solution, since each worker handles its own memory space and therefore requires a copy of matrix B. This is shown in the running times obtained in the tests using matrixes of 8192 * 8192 elements. In the message passing solution, running time and efficiency are significantly degraded, since, due to the replication mentioned above, the memory that is available in the testing architecture becomes insufficient, swapping the required structures to disc and thus significantly degrading algorithm performance.

In the future, the behavior with even larger matrix sizes will be studied, together with other parallelization strategies that mainly avoid data replication.

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] Andrews G. "Foundations of Multithreaded, Parallel and Distributed Programming". Addison Wesley Higher Education 2000. ISBN-13: 9780201357523 .

[4] Grama A., Gupta A., Karpis G., Kumar V. "Introduction to Parallel Computing". Pearson – Addison Wesley 2003. ISBN: 0201648652. Second Edition (Chapter 3).

[5] <https://computing.llnl.gov/tutorials/pthreads> (2010)

[6] <http://www.open-mpi.org> (2010)

[7] 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.

[8] Leopold C., "Parallel and Distributed Computing. A Survey of Models, Paradigms and Approaches". Wiley, 2001. ISBN: 0471358312 (Chapters 1, 2 and 3).

[9] 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.

[10] HP, "HP BladeSystem". <http://h18004.www1.hp.com/products/blades/components/c-class.html>. (2011).

[11] HP, "HP BladeSystem c-Class architecture". <http://h20000.www2.hp.com/bc/docs/support/SupportManual/c00810839/c00810839.pdf>. (2011).

Distributed Search on Large NoSQL Databases

Fernando G. Tinetti¹, Francisco Paez, Luis I. Aita, Demian Barry

III-LIDI, Facultad de Informática, UNLP, La Plata, Argentina

Fac. de Ingeniería, UNPSJB, Sede Pto. Madryn, Puerto Madryn, Argentina

¹Investigador Comisión de Investigaciones Científicas de la Prov. de Bs. As.

Abstract - *This work focuses on performance and scalability of different policies for solving queries on large noSQL databases with clusters. Distribution of data and queries are amongst the main problems, given the distributed nature of clusters: basically a set of networked computers. The basic centralized model (for both, data and processing) is used as a departure point and different distributed configurations are experimented with, in order to determine several guidelines for performance improvement. Apache Solr has been used for database management and search server. The current contents of the Wikipedia in Spanish (with about 4.5 GB) have been used as an example of a NoSQL database for experimentation.*

Keywords: Parallel and Distributed Computing, Distributed Search, Data Sharding, Mapreduce, Apache Solr.

1 Introduction

Currently, there are large amount of websites with large amount of data available, which is necessary to handle in an efficient way. Several reasons for the information volume growth have been [1] [11] [13] [7]:

- The popularity of content management systems (CMS, Content Management Systems) as portals general and as platforms for collaboration in particular.
- The so called Web 2.0, roughly defined as the current set of applications with high levels of interaction and access to multimedia data.
- The data generated within organizations, either as output or intermediate process of production systems or by digitizing existing documents.

In summary, there has been an exponential growth in volumes of information produced which, in turn, implies handling terabytes and petabytes of information instead of gigabytes. This scenario has led to the challenge of improving the so called information retrieval search tools using different/new techniques.

Scalability, availability, and performance in handling large volumes of information are now mandatory for most applications in this context, usually requiring techniques of distributed systems. Some of the techniques presented in this work include: load balancing, replication, and horizontal distribution (sharding) of information [1] [8]. The White House has used a combination of Drupal and Apache Solr in

its portal of document access/contents [15] [12]. In general, solutions to this problem must include strategies for scalability, availability, and performance.

1.1 Information Retrieval in Large Volumes of Data

Sequential search has to be discarded, given its lack of scalability. Some auxiliary data structures are necessary, that allow quick searches. Indexing provides data structures that facilitate information searching and retrieval quickly and accurately. Some indexing examples are: inverted index [4], a citation index, a matrix or a tree [9] [6] [2].

The indexing process usually requires analysis and processing of documents to include in the index: stemming, tokenization, phonetic analysis, etc. These steps introduce important issues and challenges for processing [4] [8], which are beyond the scope of this work. Instead, this work is focused on alternatives for distributing the indexes and queries in a heterogeneous and scalable environment. A set of desirable properties of a feasible solution should be related to [1]: performance and heterogeneous data, fault tolerance, and heterogeneous hardware platforms.

Performance and Heterogeneous Data: traditional databases (usually called relational or SQL databases) have a lot of effort on the issue of performance where most techniques take advantage of parallelization and partitioning of information, relying on structured data stored in (relational) tables. The problem is different with heterogeneous information, not just the information stored in the database but also all the surrounding information such as documents, pictures, videos, sound, mail, etc. Heterogeneous information causes an increase in the volume of stored data which requires restructuring and rethinking the forms of storage. Also, heterogeneous information fundamentally requires some at least some restructuring of the way in which information must be retrieved. It is worth noting that information retrieval usually requires knowledge of meaning and understanding of the data to retrieve.

Fault Tolerance: one of the effects of distributing information in order to increase performance in information retrieval is that the distributed system must consider fault tolerance. The retrieved information must be consistent, even when one some of the nodes become unavailable. Even when handling failures is beyond the scope of this paper, transactions of a database should have the so-called ACID

(Atomicity, Consistency, Isolation, Durability) property or properties. This paper focuses exclusively on the consistency and handling of distributed information indexing and recovery.

Heterogeneous Hardware Platforms: in order to guarantee good performance in information retrieval it should be possible to increase the number of participating nodes in a search. Traditional databases parallelization are usually focused on homogenous hardware, this limiting the growth in number of nodes. Moreover, noSQL solutions for managing large volumes of information are usually based on a set of heterogeneous computing nodes. There are various techniques for configuring heterogeneous environments which at least will be discussed in this paper.

2 Techniques on NoSQL Databases

Several common techniques are applied in current NoSQL databases: indexing on “shards”, shared nothing data distribution, data replication for load balancing, scatter and gather on distributed data, and map/reduce processing. These techniques are briefly explained below.

Indexing on Shards: basically, sharding is a process similar to that of horizontal partitioning of data in a standard (structured) SQL database [5]. Sharding provides multiple capabilities for scaling, allowing to divide data and indexes on multiple servers which are known as shards. Indexing shards is the process of producing a data structure that facilitates searching and retrieving some kind of information from data in its original form [4]. Common generated data structures are inverted index, index of pointers, matrix, or tree. The indexing process usually requires analysis and processing of documents to include in the index: stemming, tokenization, phonetic analysis, etc. These steps introduce important issues and challenges at processing [4], which are beyond the scope of this work. Every query has to be processed in every shard, and finally a single response is built as an aggregate result of individual shard results. This technique specially suited on large volume of data. Database sharding is directly related to the shared nothing data distribution.

Shared Nothing Data Distribution: shared nothing focuses independence of nodes, distribution of information and processing. A shard is a shared nothing node which handles a set of documents indexed by any criteria. Also, a shard has its own mechanisms for ranking, sorting, and retrieval of information, depending on information or application needs. Possible data distributions can be thematic, ontological, segmented according to preferences, or even combinations of them. In all cases, techniques can be combined with traditional databases, such as replication and parallelization on shared disk (a traditional cluster with a storage area network) [10]. In general, the concept of Shared Nothing ensures some information consistency, but it is not necessarily ACID compliant. Also, these distributions make easier using independent heterogeneous nodes with their own memory unit, disk storage and processing. Nodes are

necessarily interconnected by a network and, clearly, the architecture requires extra effort in coordination and synchronization. Data replication for load balancing, scatter and gather on distributed data, and map/reduce processing are some techniques used for coordinating the shared nothing nodes.

Data Replication for Load Balancing: the architecture must guarantee a set of nodes with consistently replicated information across all nodes. The search engine has a pool of data nodes in which the information is searched. Queries are not parallelized, but distributed between nodes, which are independent and capable of solving queries on local data. Data recovery is done in the nodes, and distribution is done at the load balancer. This strategy neither solves the space problem nor parallelizes search [14].

Scatter and Gather on Distributed Data: this method is used when data are not replicated, where the query is broadcasted from a coordinator to every node known to have data. Then, each node processes and sends a reply with the information locally found. All replies are processed in the coordinator which, in turn, consolidates into one consistent reply to the request source. An additional advantage of the method is that data nodes may additionally distribute the data into other new nodes. A hierarchical distribution is then constructed, which is not visible to the “overall” coordinator. In general, the information is partitioned among nodes and queries are effectively parallelized. However, there are also disadvantages: distribution overhead specifically with logical segmentation and some query overhead in the coordinator, which has to generate the query result from gathered data. Some logical segmentation/s, such as the ontological requires knowledge and information about the contents (data) to be stored. In some cases this knowledge is relatively complex to obtain, especially with the implementation of ontological rules [14]. Map/reduce is an effectively used technique in this context.

Map/Reduce Processing: traditional databases (usually called relational or SQL databases) have included a lot of effort on the issue of performance where most techniques take advantage of parallelization and partitioning of information, relying on structure. Usually, NoSQL databases start from text and/or heterogeneous and not necessarily structured data. Map /reduce is a good technique for processing a large volume of data in parallel. The model provides a mechanism for data partitioning that can make a “smart” distribute according to predefined rules on self-contained different nodes. An additional advantage lies in saving space in the result of shared keys by reducing them within a document [3].

3 Experimentation Guidelines

The work in paper is focused on verifying the effectiveness of a NoSQL database manager as a model for scalability and efficiency in information retrieval. The specific chosen manager is the Apache Solr implementation of Apache

Lucene. Apache Solr has several advantages for this analysis, since:

- Apache Solr is freely available.
- Apache Solr allows testing heterogeneous document indexing.
- Allows future tests and other analyses with similar platforms of the same family.

There are several performance indices which can be measured by experimentation: CPU load, average response time to queries, memory and swap usage, disk accesses, and network bandwidth involved, among others. The two most important indices have shown to be CPU load and average response time, since in some way include (are more or less directly related to) the other indices. Performance and state is measured in both server/s and clients, thus obtaining an approximation of the distributed client/server system state as well as the state of individual computers and processes. At a higher level, the analysis is focused on performance scaling as well as quality of service/individual query response time.

The full backup of the current Wikipedia articles in Spanish has been used as a real environment of documents to be indexed and searched, with approximately 1800000 items (4 GB of data on disk). Several tests were designed in order to measure the performance of different server configurations. Every test involves the simulation of several concurrent user processes and multiple specific queries with the following characteristics:

- The whole set of words in the dictionary of the Spanish Royal Academy of approximately 86,000 words was used.
- Each query was generated by random grouping from 2 to 4 words. This ensures randomness and heterogeneity of queries.
- Queries are issued from several concurrent client processes (simulated by runtime threads). The numbers of threads used were 64, 128, 256, 384, 512, 768 and 1024, thus allowing a progressive analysis of workload/requirements.
- Every test is repeated 10 times in order to obtain the corresponding average of each index measurements.

The open source tool Siege [16] was used for generating and measuring the concurrent client's environment of each experiment, and awk was used for constructing each specific query.

It is worth noting that database update is not taken into account in these experiments, since only the number of query (recovering information) requests is being considered from concurrent clients. There are not delete/change requests which would change database content/s.

The experiments were also carried out with three server configurations: a centralized server, a server with two shards and a replicated server. Fig. 1 shows the centralized server configuration, which is standard, and used in this work for comparison, in order to have a reference point. Fig. 2 shows the server configured with two shards (both shards have the same number of documents). The server front-end has a minimum workload: query replication to both shard servers

and aggregation of results from both shard servers in order to send a unique result to each query. Fig. 3 shows the specific configuration defined for a replicated server with two replicas. The Master Server originally indexes the documents and manages replicas. Every replica independently handles its own queries, and the server front-end has a low workload: balances queries (round-robin) and sends results to the proper client. Even when the Master Server has to deal with new documents and their indexing, most of the problem is still found at the query problem and the workload generated by multiple concurrent client processes.

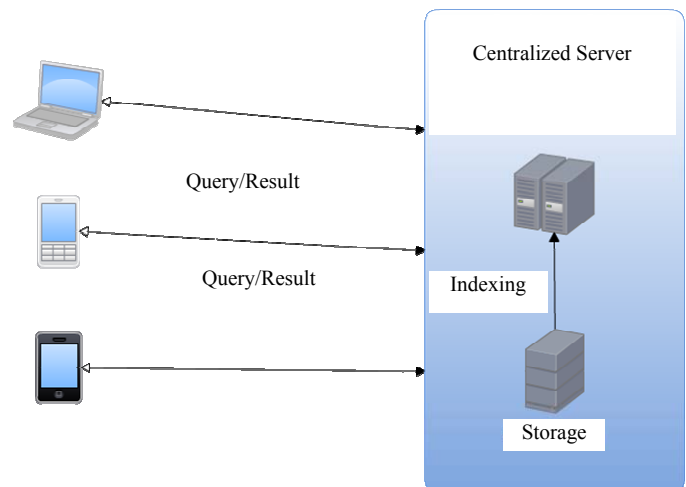


Figure 1: Centralized Server.

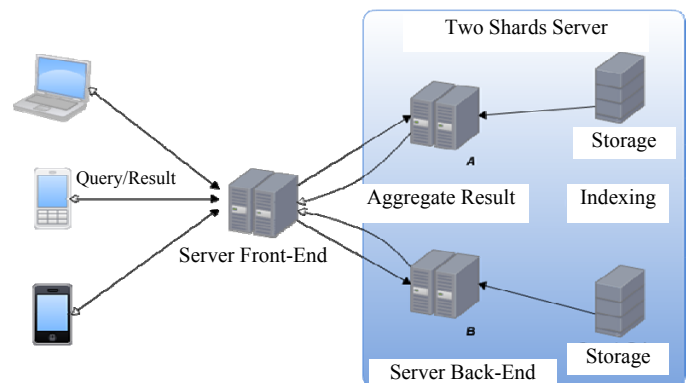


Figure 2: Two Shards Server.

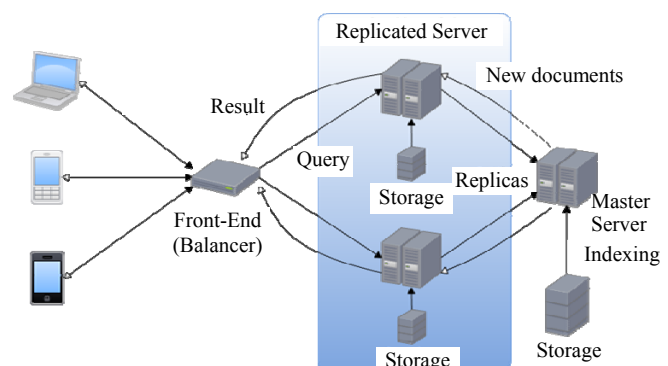


Figure 3: Replicated Server with Two Replicas.

The sar tool is used in every server configuration to monitor and account the selected performance indices. Experiments are triggered at the server side, synchronized with the sar command at the server side. Each performance index sample is taken every two seconds at experiment runtime and stored for later processing. In case of several servers, indices are aggregated and processed after running the experiments.

4 Results

Several combinations of hardware and software were used in order to test the server configurations and client requests. The main characteristics of the computer from which the script that collects all the client-side information and performs all the requests to Solr server/s are:

- AMD Athlon(tm) 64 X2 Dual Core Processor 5200+
- 1 GB DDR2 RAM
- 100 Mb/s Ethernet NIC
- Fedora Core 11 64 bits

Two different computers were used for the server side, so that heterogeneous hardware can be measured and evaluated. The most powerful computer/hardware is used for the stand alone/centralized server (Fig. 1 above) as well as one of the shard and one of the replica servers (Fig. 2 and Fig. 3 above). The main characteristics of this computer (the most powerful at the server side) are:

- Intel i3 CPU 540 @ 3.07GHz
- 8 GB DDR3 RAM
- 1000 Mb/s Ethernet NIC
- Ubuntu Server 10.04 LTS 64 bits

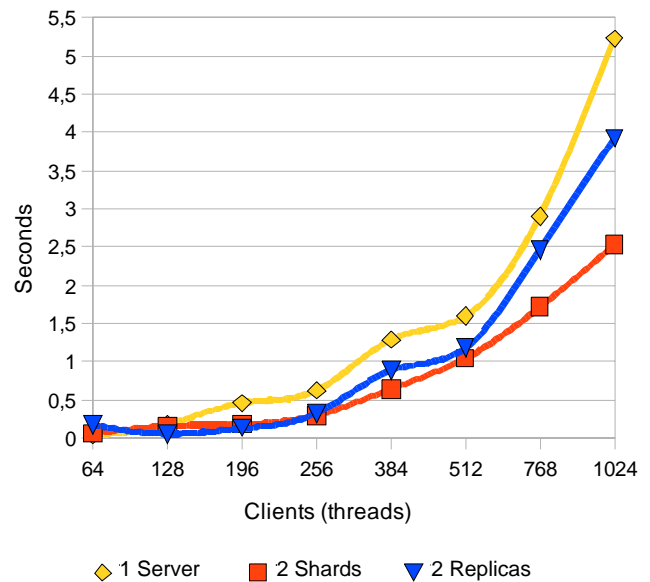
The less powerful computer used at the server side has the following characteristics:

- Intel Core 2 Duo CPU E7400 @ 2.80GHz
- 2 Gb DDR2 RAM
- 1000 Mb/s Ethernet NIC
- Ubuntu Server 10.04 LTS 64 bits

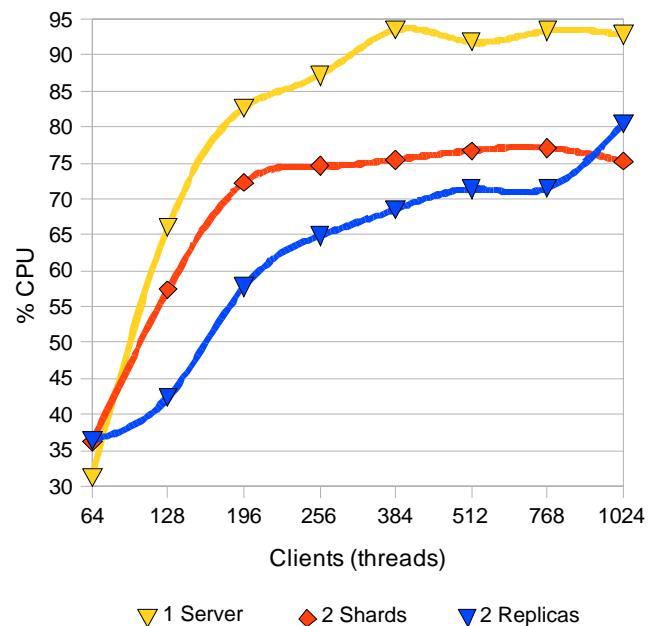
Apache Tomcat is used in every server configuration with the default installation; the only changed value is the number of threads, raised to 1024. The JVM (Java Virtual Machine) running Apache Tomcat was specifically configured so that the server is monitored by enabling JMX (Java Management eXtensions). Apache Solr is used almost with the default configuration and installation from binaries, specifically defining the Wikipedia in Spanish documents to be indexed and searched for specific data.

Fig. 4 shows the performance in seconds for different number of clients triggering queries for the three defined server configurations. Even when all the configurations have a performance degradation starting at the 512 clients, the shards configuration has a near linear increase in the time which, in turn, suggests a better scalability at least in terms of number of requests. As expected, the stand-alone server configuration (shown as "1 Server" results) has performance degradation for lower number of concurrent clients: between 256 and 384 and, also, degradation is far from being linear

starting at 512 clients. As shown on Fig. 4, the replicated server has an intermediate behavior in performance between that of the stand-alone server and the sharded server. Perhaps the worst replicated server behavior characteristic is that shown starting at 512 clients, since performance degradation is far from being linear, even worse than that for the stand-alone server. This, in turn, shows that replication servers should be very carefully designed, configured and monitored in order to avoid hot spots and/or high performance degradations under requests stress.

**Figure 4:** Response Time Performance.

In Fig. 5 the CPU usage is shown for the experiments already shown in Fig. 4. The stand-alone server configuration

**Figure 4:** Average Server CPU Usage.

is almost overloaded starting at 85 concurrent clients, with more than 85% CPU usage. Sharded and replicated server configurations are not necessarily lightly loaded (both are above 65% CPU usage starting at 256 concurrent clients), but do not get overloaded, the servers are always below 80% CPU usage. The CPU usage explains almost directly the reason for the performance obtained with each server configuration, since results are almost directly proportional. The direct relationship Response Time - CPU usage allows to exclude the analysis on other important factors such as network load, disk accesses, RAM usage/footprint, etc.

Since heterogeneous computers are used at the server side, further analysis would be useful in order to explain results as well as define better balancing and scaling load strategies. Fig. 5 shows the CPU usage per each computer at the server side. Clearly, the stand-alone server CPU usage (shown as "1 Server") is the same as that shown in Fig. 4. Results shown as "S1" correspond to the sharded server configuration on the best computer and those shown as "S2" to the sharded server on the worst computer. Clearly, the worst computer is almost always overloaded starting at 196 concurrent clients. The best computer used at the server side is almost always lightly loaded. There is a huge unbalanced workload for heterogeneous servers, and it seems to be a clear index for workload balance: CPU usage.

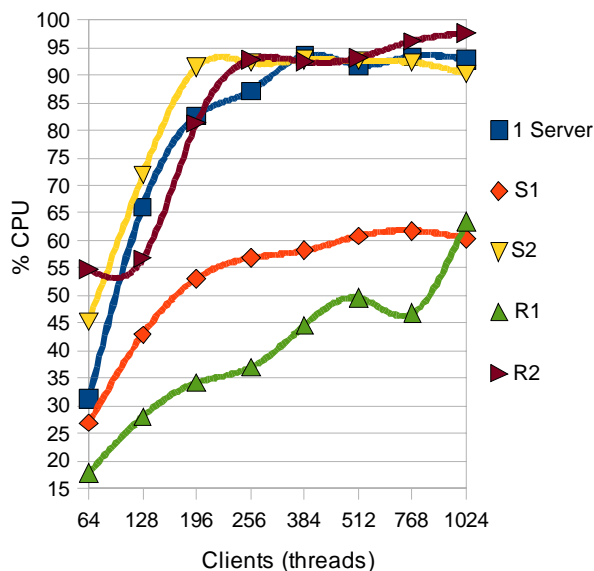


Figure 5: CPU Usage per Server Computer.

Results shown in Fig. 5 for the replicated servers (R1 and R2, respectively) are analogous to those explained for the sharded servers.

In general, analyzing data shown in Fig. 3, Fig. 4, and Fig. 5, several interesting remarks can be done

- The sharded server configuration almost always provides the best performance.
- Distributing data and/or queries at the server side almost always improves performance.

- There is still work to be done for enhancing workload balance, which could further improve performance, specifically with heterogeneous hardware.
- Even when some of the distributed server computer/s are overloaded, the average CPU usage is directly related to performance, i.e. overloading does not necessarily imply too high performance penalties provided there are not overloaded server computers which are able to handle incoming requests.

5 Conclusions and Further Work

This paper has shown several configuration guidelines and obtained results for NoSQL databases. Experiments with heterogeneous hardware are also included, basically as a *proof of concept* and, also, for further analysis of specific results. Working with heterogeneous data can be considered as granted on NoSQL databases by its own nature. Infrastructure software such as Apache Solr has proven to be successful not only for starting a (*server side*) content management system, but also to experiment and measure runtime tests. Specific experimentation and measurement on a distributed system imply using tools and methodologies on the client side, and a combination of Siege and awk has been used for generating representative requests load on different server configurations.

Several tests have shown that a completely distributed data and information recovery configuration by defining shards provides the best runtime results.

There are several immediate tests and hardware and software configurations to experiment with:

- Using more servers for shards as well as for replicas, also with more heterogeneity.
- Fine-tuning of data and workload on each heterogeneous server.
- Combinations of sharding and replication, since those options do not exclude each other.

Other research lines are not so immediate, since require more analysis and experimentation, among other tasks. A possible step forward is the heterogeneous distribution of indexes according to different criteria, such as server heterogeneity and type of query (e.g. combination of words).

6 References

- [1] Azza Abouzeid, Kamil BajdaPawlikowski, Daniel Abadi1, Avi Silberschatz, Alexander Rasin, "HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads", Proceedings of the VLDB Endowment (2009), Vol. 2, Issue: 1, pp. 922–933.
- [2] J. Chris Anderson, Jan Lehnardt, Noah Slater, CouchDB: The Definitive Guide, O'Reilly Media, Jan. 2010, ISBN 1449379680.
- [3] Jeffrey Dean, Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Communica-

- tions of the ACM - 50th anniversary issue: 1958 - 2008, Vol. 51, Issue 1, Jan. 2008.
- [4] Erik Hatcher, Otis Gospodnetić, Lucene in Action, 2nd ed, Manning Publications Co., 2004.
- [5] Cal Henderson, "Building Scalable Web Sites", O'Reilly Media, 2006.
- [6] Eben Hewitt, Cassandra: The Definitive Guide, O'Reilly Media, Nov. 2010, ISBN 1449390412.
- [7] Curt Monash. The 1-petabyte barrier is crumbling, Networkworld, Aug. 2008 <http://www.networkworld.com/community/node/31439>.
- [8] Ken North, "The NoSQL Alternative, Low-cost, high-performance database options make gains," Information Week, May 2010.
- [9] Eelco Plugge, Tim Hawkins, Peter Membrey, The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing, Apress, October 2010, ISBN 1430230517.
- [10] Michael Stonebraker, "The Case for Shared Nothing", Database Engineering, Vol. 9, No. 1, 1986, <http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf>
- [11] Carl W. Olofson, Worldwide RDBMS 2005 vendor shares, Technical Report 201692, IDC, May 2006.
- [12] Thoughts on the Whitehouse.gov switch to Drupal, <http://radar.oreilly.com/2009/10/whitehouse-switch-drupal-opensource.html>
- [13] Dan Vesset, Worldwide data warehousing tools 2005 vendor shares, Technical Report 203229, IDC, August 2006.
- [14] Zhou Wei, Guillaume Pierre, Chi-Hung Chiy: CloudTPS: Scalable Transactions for Web Applications in the Cloud. Technical report IR-CS-53, Vrije Universiteit, February 2010.
- [15] WhiteHouse.gov Goes Drupal, <http://personal.democracy.com/node/15131>
- [16] Joe Dog Software, Siege, <http://www.joedog.org/index/siege-home>

SESSION

WORKSHOP ON MATHEMATICAL MODELING AND PROBLEM SOLVING, MPS

Chair(s)

Prof. Minoru Ito

An Attribute Graph Grammar for UML Package Diagrams and its Applications

Takaaki Goto¹, Tetsuro Nishino², and Kensei Tsuchida³

¹Center for Industrial and Governmental Relations, The University of Electro-Communications, Chofu, Tokyo, Japan

²Graduate School of Informatics and Engineering, The University of Electro-Communications, Chofu, Tokyo, Japan

³Faculty of Information Science and Arts, Toyo University, Kawagoe, Saitama, Japan

Abstract—Graphical representations are often used in software design and development because of their expressiveness. Unified Modeling Language (UML) for modeling in software development was proposed recently, and in 2005 it was standardized as the ISO/IEC 19501 standard.

In order to automate processing of these graphical representations using computers, a syntax for program diagrams must first be defined. We propose a framework for specifying these diagrams using a graph grammar, and for processing these diagrams automatically.

Keywords: graph grammar, UML, package diagram, SVG

1. Introduction

Graphical representations are often used in software design and development because of their expressiveness. Various graphical program description languages have been reported, including Hierarchical flowchart language (Hichart), Problem Analysis Diagrams (PAD), Hierarchical and Compact description charts (HCP), and Structured Programming Diagrams (SPD), and many Computer Aided Software Engineering (CASE) tools have been developed based on these languages [1], [2], [3].

On the other hand, the Unified Modeling Language (UML) for modeling in software development was proposed recently compared with above graphical program description languages, and in 2005 it was standardized as the ISO/IEC 19501 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 process to downstream process. In order to automate 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.

Research on graph grammars has been done by Rozenberg [4] and others. Research has also been done on UML [5] and graph grammars and graph transformations with respect to UML [6], [7], [8].

However these researches do not deal with syntax formalization for visual representation. And also graph grammars for package diagram are not proposed yet in previous researches. Therefore we provide a graph grammar for package diagram of UML to propose theoretical fundamentals of UML.

With regard to Web documents, XML and SVG have been proposed as standard document and graphical formats for the Web. Scalable Vector Graphics (SVG) [9] is a W3C Recommendation and a language for describing two-dimensional graphics and graphical applications in XML. SVG can display graphical objects on any readily available Web browser. With these formats, users can share document including graphical objects on the Web. We reported on automatic generation of SVG files and incorporated the generation method into a graphical editor for Hichart by using attribute graph grammars.

The goal of this research is to generate UML package diagrams based on a graph grammar. We propose a framework for specifying these diagrams using a graph grammar, and for processing these diagrams automatically.

2. Preliminary

2.1 Graph Grammars

Definition 1. ([4]) 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 $X \rightarrow (D, C)$ 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 . \square

An example of a production is shown in Figure 1. In the Figure, a box is a nonterminal node and a filled circle is a terminal node. X , Y , and b mean node labels and v_0 ,

v_1 , and v_2 mean node IDs. Nodes with same node label can appear in a graph, while nodes with same node ID will never appear in a graph. The production of Figure 1 indicates that after the removal of a nonterminal node with label X , embed the graph consists 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 $(a, \alpha/\beta, v_1, in)$, however this connection instruction is not described in the notation of Figure 1.

In Figure 2, the production of Figure 1 and its connection instruction are drawn simultaneously. The large box of Figure 2 indicates the left-hand side, and two nodes with label b and Y are right-hand side of the production of Figure 1.

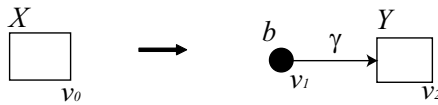


Fig. 1: An example of a production

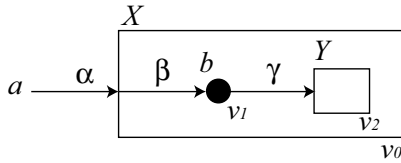


Fig. 2: An example of a production with the connection relation

An example of application of the production is shown in Figure 3. In Figure 3 $H = (V_H, E_H, \lambda_H)$ 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: $p' : X \rightarrow (D', C')$ 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 $C' = \{(a, \alpha/\beta, n_3, in)\}$.

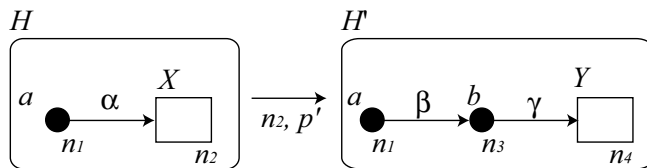


Fig. 3: An example of applying a production rule

In Figure 3, H indicates the host graph and H' is the resulting 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 2. ([10], [11]) An *Attribute edNCE Graph Grammar* is a six-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 $Syn(X_0) \cup \bigcup_{X \in Lab(D)} Inh(X)$. A semantic rule defining an attribute $a_0(X_{i_0})$ has the form $a_0(X_{i_0}) := f(a_1(X_{i_1}), \dots, a_m(X_{i_m}))$. Here f is a mapping from $V(a_1(X_{i_1})) \times \dots \times V(a_m(X_{i_m}))$ 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 . \square

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, and so on. Behavioral diagrams are used to describe the behavior of what is being modeled and include such as use-case, activity, and state-machine diagrams.

Structure 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 4 shows an example of a package diagram. The box with rectangle at the upper left indicates a package. The box with three compartments is a class. Each of three parts indicates its class name, its attribute, and its methods from top to the bottom. A plus with circle is used to represent which components the package contains. Package 1 contains Package 2 and Package3, and Package 3 contains Class2 and Class 3.

3. Graph Grammar for UML Package Diagrams

In this section we describe our Graph Grammar for Package Diagrams (GGPD), for UML package diagrams.

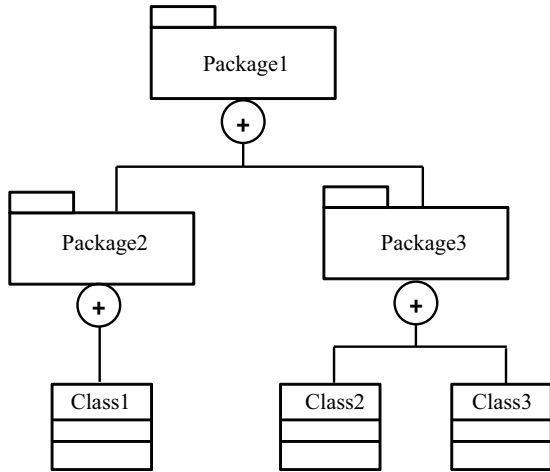


Fig. 4: An example of a package diagram

3.1 Grammar Overview

Definition 3. The Graph Grammar for Package Diagrams (GGPD), for UML package diagrams, is a six-tuple $GGPD = (\Sigma_{PD}, \Delta_{PD}, \Gamma_{PD}, \Omega_{PD}, P_{PD}, S_{PD})$. Here, $\Sigma_{PD} = \{ S, A, T, L, R, M, rop, sp, lep, rip, mip, lec, mic, ric \}$ is a finite set of node labels, $\Delta_{PD} = \{ rop, sp, lep, rip, mip, lec, mic, ric \}$ is a finite set of terminal node labels, $\Gamma_{PD} = \{ * \}$, $\Omega_{PD} = \{ * \}$, $P_{PD} = \{ P_1, \dots, P_{17} \}$ is a finite set of production rules, and $S_{PD} = \{ S \}$, is the initial non-terminal. \square

The GGPD generates package hierarchy diagrams. It is a context-free grammar and there are 17 production rules. An example of GGPD production rule is shown in Figure 5.

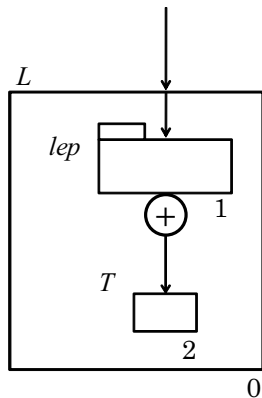


Fig. 5: An Example of a production rule of GGPD

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 capitalized label indicates a nonterminal node, and a node with uncapitalized label indicates a termi-

nal node. Our grammar generates directed graphs. However obtained graphs are drawn without arrows by assumption that the direction of each edge from top down.

3.2 Example of Derivation

Figure 6 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 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 correspond to UML package diagrams.

We can obtain a derivation tree from derivation sequence of production. Figure 7 shows the derivation tree corresponding to Figure 6. In the Figure 7, the labels show the name of production rules.

Another example of a package diagram resulting from applying the production rules is shown in Figure 8.

3.3 Generation of SVG document for package diagrams

We introduce attribute S_{SVG} which contains SVG source codes, as its value and representation corresponding to the package diagram. We have a plan to generate diagrams with animation. SVG can display on browser such as IE with SVG plugin.

SVG source codes are generated by evaluating S_{SVG} . Evaluation of attributes is performed in the bottom-up manner on derivation trees. Figure 9 illustrates the flow of generating SVG files.

Figure 10 gives examples of semantic rules with the attribute S_{SVG} .

3.4 Folding / UnFolding

When drawing package diagrams for large-scale systems, the scale of diagrams can become large, and this can make diagrams difficult to comprehend visually. This makes it necessary to process diagrams to summarize and hide information. Thus, we perform information-hiding by expressing diagrams in sentential form.

Figure 11 shows an example of a package diagram and its derivation tree before folding, and Figure 12 shows the package diagram and derivation tree after folding.

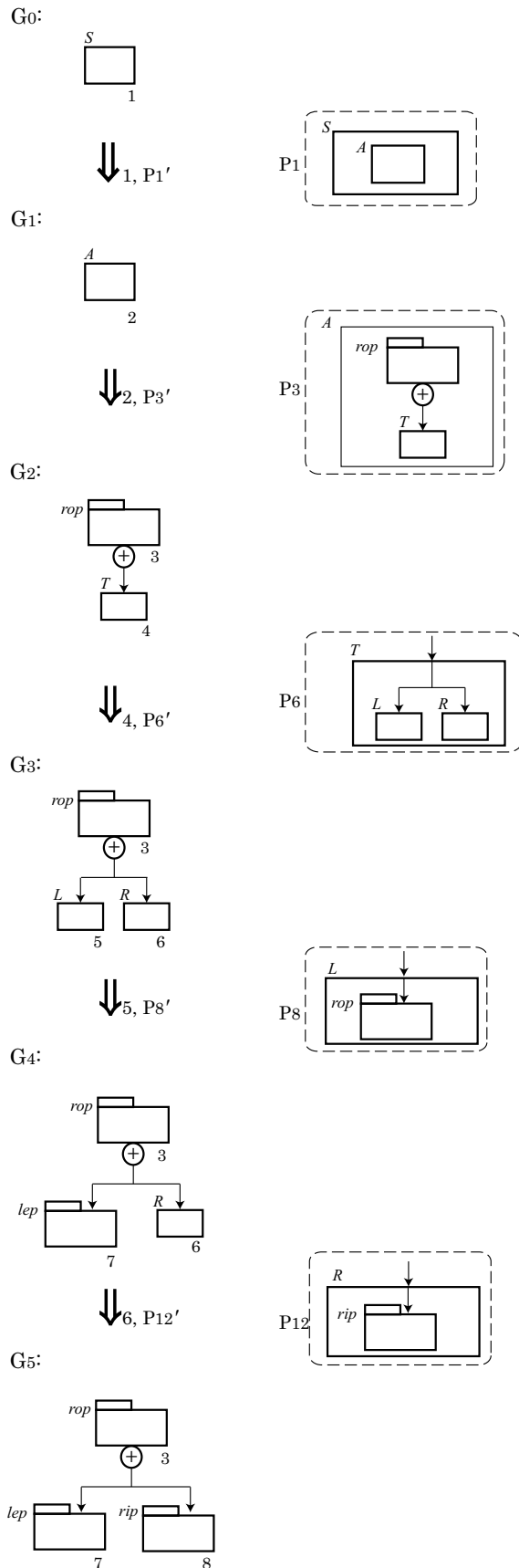


Fig. 6: An example of a GGPD derivation

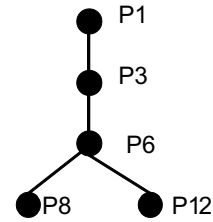


Fig. 7: A derivation tree corresponding to the tree in Figure 6

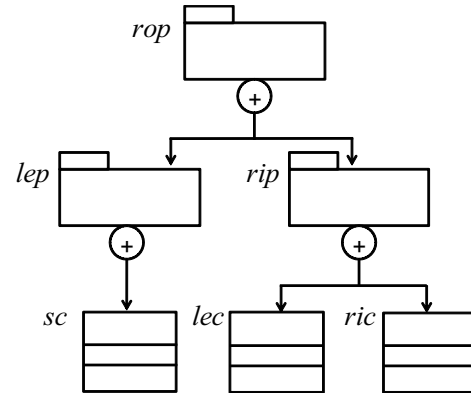


Fig. 8: An example of package diagram resulting from derivation

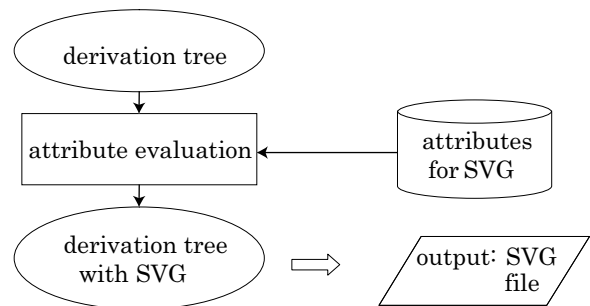


Fig. 9: Flow of generating SVG files

```

Ssvg(1) = <rect x="x(1)" y="y(1)" width="w(1)" height="h(1)"
           fill="white" fill-opacity="1" stroke="black" />...
Ssvg(0) = Ssvg(1) Ssvg(2)
    
```

Fig. 10: An examples of semantic rules with the attribute S_{SVG}

4. UML Package Diagram Editor

In this section, we explain our prototype UML package diagram editor based on the grammar described in Section 3. The editor is a syntax-directed editor and was developed in Java. Figure 13 shows a screenshot of the editor.

On the editor, when a non-terminal node displayed on the

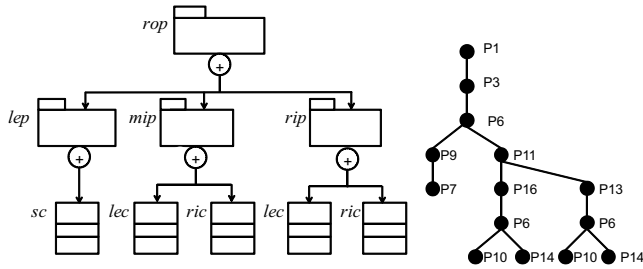


Fig. 11: A package diagram and its derivation tree before folding

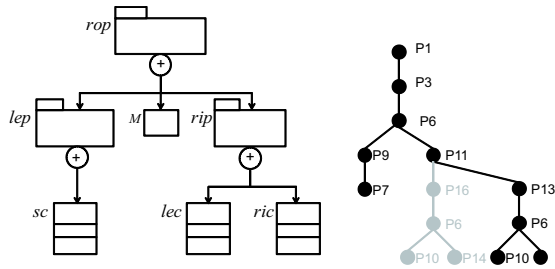


Fig. 12: A package diagram and its derivation tree after folding

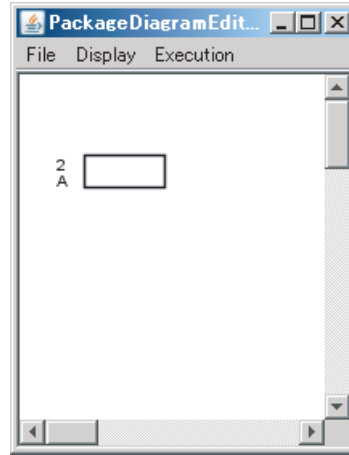


Fig. 14: A nonterminal node on the editor

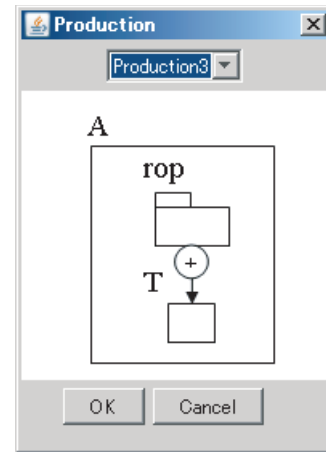


Fig. 15: The production rule display screen of the package diagram editor

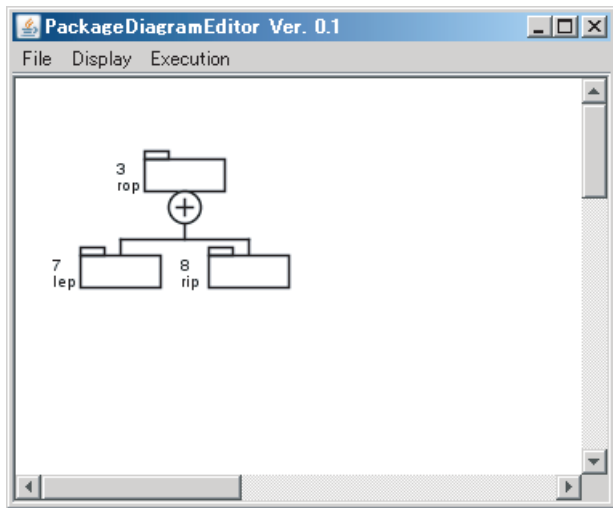


Fig. 13: A screenshot of the package diagram editor

editor screen is selected, a screen displaying the production rules that can be applied to the non-terminal node is displayed. Figure 14, 15, and 16 show a screen shot when the non-terminal node with node ID of 2 and labeled A in the package diagram editor screen is clicked (Figure 14), and the applicable production rules are displayed (Figure 15). After choosing a production rule, the production rule is applied to the non-terminal node (Figure 16).

The applied production rules can also be displayed as a derivation tree, as shown in Figure 17.

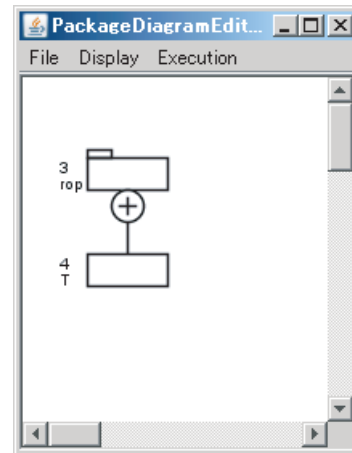


Fig. 16: An example of applying production rule on the editor

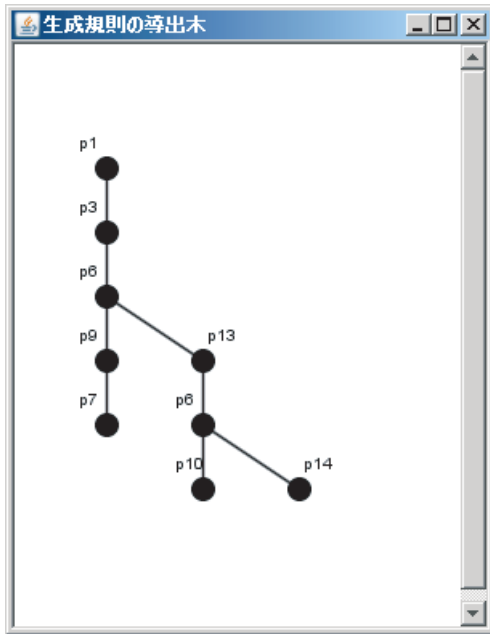


Fig. 17: The derivation tree display screen of the package diagram editor

When users execute an editor command, SVG files can be automatically generated by evaluating SVG attributes. The evaluation is executed by traversing on the derivation tree. Figure 18 is an example of the display of a package diagram in SVG.

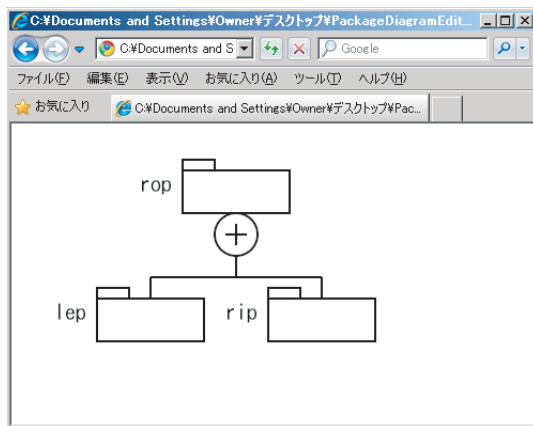


Fig. 18: An example of the display of a package diagram in SVG

5. Conclusion

In this paper, we have defined a graph grammar for generating the hierarchical structure of UML package diagrams. We have also created a syntax-directed diagram editor for the defined grammar. A future issue for study is to implement

syntactic analysis. The editor developed here is able to generate diagrams according to the grammar and complying with the syntax through human intervention, but it is not able to determine, from an arbitrary input, whether a diagram conforms or not. By implementing syntactic analysis, automatic processing of arbitrary input diagrams will be possible. Application of this technology to automatic generation of software documentation is another possibility.

References

- [1] K. Harada, *Structure Editor*. Kyoritsu Shuppan, 1987, (in Japanese).
- [2] Yoshihiro Adachi, Youzou Miyadera, Kimio Sugita, Kensei Tsuchida, and Takeo Yaku, "A Visual Programming Environment Based on Graph Grammars and Tidy Graph Drawing," in *Proceedings of The 20th International Conference on Software Engineering (ICSE '98)*, vol. 2, 1998, pp. 74–79.
- [3] Takaaki Goto, Kenji Ruise and Takeo Yaku and Kensei Tsuchida, "Visual Software Development Environment Based on Graph Grammars," *IEICE Transactions on Information and Systems*, vol. 92, no. 3, pp. 401–412, 2009.
- [4] G. Rozenberg, *Handbook of Graph Grammar and Computing by Graph Transformation Volume 1*. World Scientific Publishing, 1997.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] D. Petriu and H. Shen, "Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications," in *Computer Performance Evaluation: Modelling Techniques and Tools*, ser. Lecture Notes in Computer Science, vol. 2324, 2002, pp. 183–204.
- [9] W3C Web Site. Scalable Vector Graphics (SVG), <http://www.w3.org/TR/SVG/>.
- [10] 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.
- [11] 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.

Classification of Idiopathic Interstitial Pneumonia CT Images using Convolutional-net with Sparse Feature Extractors

Taiju INAGAKI¹, Hayaru SHOUNO¹, Shoji KIDO²

¹ Graduate School of Informatics and Engineering,

University of Electro-Communications,

Chofugaoka 1-5-1, Chofu, Tokyo JAPAN ² Applied Medical Engineering Science,

Graduate School of Medicine, Yamaguchi University,

Tokiwadai 2-16-1, Ube, Yamaguchi JAPAN

Abstract— We propose a computer aided diagnosis (CAD) system for classification of idiopathic interstitial pneumonias (IIPs). High resolution computed tomography (HRCT) images are considered as effective for diagnosis of IIPs. Our proposed CAD system is based on the convolutional-net that is bio-plausible neural network model inspired from the visual system such like human. The convolutional-net extract local features and integrate them in the process of hierarchical neural network system. For natural image recognition by convolutional-net, Gabor feature extraction is known to give a good performance, however, the HRCT images may have different properties from those of natural images. Thus, we introduce a learning type feature extraction called “sparse coding” into the convolutional-net, and evaluate performance for classification of IIPs.

Keywords: Computer Aided Diagnosis, Idiopathic Interstitial Pneumonia classification, Convolutional-net, Sparse coding

1. Introduction

In the field of medical image diagnosis using high resolution computed tomography (HRCT) is effective for classifying of idiopathic interstitial pneumonias (IIPs). Using the HRCT image, we may observe the site of IIPs is diffused in the lung, however, determining the border of the disease site is difficult work, and the IIPs on HRCT images shows a lot of varieties in patterns. Thus, the quality of diagnosis is influenced by the ability of diagnostician, and improving the quality is desired for proper treatment. The second opinion system, which means plural diagnosticians opinions are taken into consideration for diagnosis, is an answer for the problem. However, this system makes the diagnosticians diagnose over twice patients, that is the second opinion system might be burden for diagnosticians. Moreover, because of the large number of variations in image

pattern of IIPs, a lot of cost may require to educate for a skilled diagnostician. Hence, the diagnosis aid system using computer is desired for objective diagnosis in these decades. The computer aimed diagnosis (CAD) system is designed to provide a second opinion using computer analysis from the obtained images, and we can consider many types of CAD systems. In this study, we try to construct a computer diagnosis aid using convolutional-net, which is a kind of artificial neural network inspired from the visual system of human [1][2][3][4]. Roughly speaking, the mechanism of the convolutional-net consists of two components: one is the local feature extraction, and the other is integration of the extracted features with non-linear modulation. The feature extractor components called S-cells, which comes from simple cell in the visual area of brain, respond to the similarity between the input and the preferred feature of the cell. The integration components called C-cells, which comes from complex cell in the brain, integrate the extracted feature by spatial pooling of the S-cell outputs. We assume each type cell arranged in the 2-dimensional lattice called cell-plane and cell in the identical cell-plane have same properties. By this assumption, cell in an identical plane could share the weight of the connection. The mathematical notation of the weight sum sharing for the input can be described as a convolution, so that this type of network is called “convolutional-net”. To determine the preferred feature of S-cells, we introduce a learning rule called “sparse coding” [5]. The sparse-coding assumes any input as a weighted sum of linear bases, and the bases are determined to satisfy that as much as the weights for bases should take 0 value for whole input data with compensation for the overcomplete bases. Olshausen & Field show that applying the sparse coding to the small part called image patch of natural scene, they obtained Gabor feature like preferred bases, which is usually used to denote the property of the simple cell [5]. We apply the sparse-coding bases into the

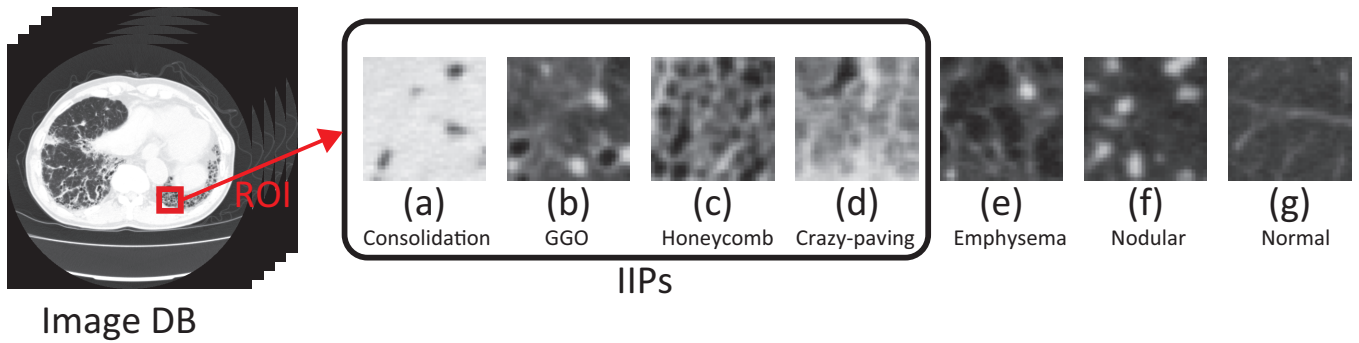


Figure 1: Typical CT images of diffuse lung diseases: The top row shows each overview, and bottom shows magnified part (ROI) of each lesion. From (a) to (g) represents “Consolidation”, “GGO”, “Honeycomb”, “Crazy-Paving”, “Nodular” “Emphysema”, and “Normal” image respectively.

feature extractor weight of the convolutional-net.

For IIPs classification, several approach are proposed, and, in recent years, a “texton” base system are focused in classification of lung diseases[6]. A texton means the clustered features from the collection of small patch of images, and texton base system use the collection of similarities between an input and each texton as a feature vector. Thus, our approach can be regarded as an extension of this texton base approach.

In this study, we developed a prototype CAD system for classifying IIPs. Our CAD system take a segmented image which is taken from the HRCT image of lungs, and classify the input image into following named classes, that is, consolidation, ground-grass opacity (GGO), honeycomb, crazy-paving, nodular, emphysema and normal classes. The lesion of this disease is spread in lung, and has a lot of image patterns even in the same class. Fig.1 shows a typical image example of each disease HRCT image. The left shows an overview of the axial HRCT images of lungs including lesion, and the right shows segmented images of typical examples of lesion from the left image collections. The consolidation and GGO patterns are often appeared with the cryptogenic organizing pneumonia diseases (COPD). The GGO pattern is also often appeared in the non-specific interstitial pneumonia (NSIP). The crazy-paving pattern have reticular pattern with partial GGO patterns, which appeared in also NSIP. The honeycomb pattern has more rough mesh structure rather than that of the crazy-paving, and it appeared in idiopathic pulmonary fibrosis (IPF) or usual interstitial pneumonia (UIP).

2. Method

In this section, we explain about more detailed convolutional-net formulation and learning method of sparse

coding using in our CAD system.

2.1 Structure of Convolutional- net

The convolutional-net mainly consists of two types of cells. One is called “S-cell” which is used for feature extractor. The S-cell have local connection window called receptive field, and the local connection weight dictate preference of the S-cell, so that the local connection weight is sometimes called preferred vector. When an input is appeared to the receptive field, the S-cell calculates a similarity between the input and the preferred vector for responding. The other type of cell is called “C-cell” which is used for reduction of local input pattern deformation, such that shift, rotation, and so on. The C-cell calculates spatial pooling of the S-cell that have same preferred vector in the local area. This spatial pooling calculation is sometimes called “blurring” or “sub-sampling” [1][3]. These calculation manners are originally proposed by Hubel & Wiesel [7].

We treat each type of cell as arranged in 2-dimensional lattice called “cell plane”. The cell in a cell plane has same preferred vector except the receptive field position, which is just differ as the position of the cell in the cell plane. Introducing the cell plane structure, we can treat the connection between the cell planes as the convolution. Thus, we call this type of network as “convolutional-net”. In the center part of the Fig.2 shows a schematic diagram of the convolutional-net. Each rectangle in the part shows cell plane that includes same type of cells, and whole cells have only local connections.

As in mathematical form, we denote the response of the S-cell at the location \mathbf{x} in the k -th cell plane as $u_s(\mathbf{x}, k)$,

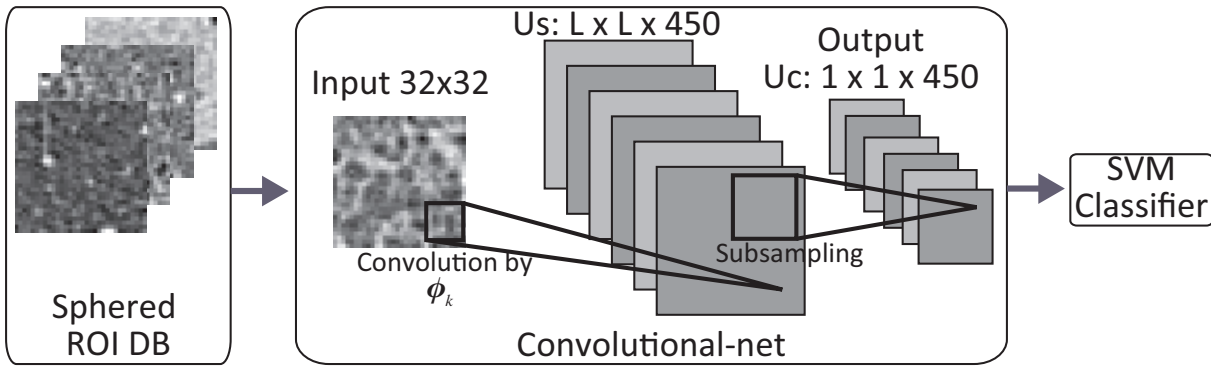


Figure 2: Schematic diagram of our CAD system using convolutional-net. In the convolutional-net part, each rectangle shows cell plane which includes same type of cells arranged in the 2D array.

and denote it as a convolution form:

$$u_s(\mathbf{x}, k) = \varphi \left[\frac{\sum_{\nu} \phi_k(\nu) I(\mathbf{x} + \nu)}{\sqrt{\sum_{\nu} \phi_k(\nu)^2} \sqrt{\sum_{\nu} I(\mathbf{x} + \nu)^2}} - \theta_k \right], \quad (1)$$

where $\varphi[\cdot]$ means the half-wave rectified function:

$$\varphi[s] = \begin{cases} s & \text{if } s > 0 \\ 0 & \text{else} \end{cases}, \quad (2)$$

θ_k means threshold value for the cell in the k -th cell plane, and $\phi_k(\nu)$ means the connection weight for the relative location to \mathbf{x} . Introducing a vector notation for index of receptive field ν , that is ϕ_k as $\phi_k(\nu)$, \mathbf{I}_x as $I(\mathbf{x} + \nu)$, we can denote eq.(1) as:

$$u_s(\mathbf{x}, k) = \varphi \left[\frac{\phi_k \cdot \mathbf{I}_x}{\|\phi_k\| \|\mathbf{I}_x\|} - \theta \right] \quad (3)$$

where dot operator in the numerator means the inner product of vectors, so that the first term in the function $\varphi[\cdot]$ means the similarity in the meaning of direction cosine. Thus, we can interpret the eq.(1) as two step calculation, that is the first step is calculation of similarity between local input \mathbf{I}_x and the preferred vector ϕ_k , and the second is modulate the similarity by the threshold and half-wave rectification.

The C-cell function also denote as a convolution for the spatial pooling in the S-cell plane:

$$u_c(\mathbf{x}, k) = \psi \left[\sum_{\xi} \rho(\xi) u_s(\mathbf{x} + \xi, k) \right], \quad (4)$$

where ξ indicates the connection location relative to the \mathbf{x} , $\rho(\xi)$ means the connection weight, and $\psi[\cdot]$ means the modulation function. In this study, to keep the network structure simple, we adopt following conditions. We assume connection between $u_c(\mathbf{x}, k)$ have whole spatial pooling for $u_s(\mathbf{x}, k)$ which means $u_c(\mathbf{x}, k)$ denote as a single unit $u_c(k)$ and it have full connection to the whole units in the previous

plane $u_s(\mathbf{x}, k)$. Moreover, we also assume whole connection weight as homogeneous, that is $\rho(\xi) = 1$, and modulation function $\psi[\cdot]$ as linear modulation function $\psi[u] = u$. Hence, we can denote the C-cell for the k -th feature described in eq.(4) as:

$$u_c(k) = \sum_{\xi} u_s(\xi, k). \quad (5)$$

Now, we can consider the output of the convolutional-net $u_c(k)$ for the input $I(\mathbf{x})$ as a kind of the conversion from the input to a feature vector, so that we should classify the feature vector into the class category. In order to classify $u_c(k)$, we introduce a support vector machine (SVM), which is developed in the field of machine learning, for classification in the next stage [8].

2.2 Learning of Preferred Feature by Sparse Coding

For applying a convolutional-net into the natural image understanding, Gabor filters is usually adopted in the feature extractor connection ϕ_k . The Gabor filter is suitable for extraction of line or edge segment in the image, and those feature components are considered important in the field of natural scene understanding [3] [1]. However, it is doubtful that line or edge components in the segmented image of the IIPs is effective to the classification. Thus, we introduce learning base algorithm called sparse coding to determine the feature extraction vector set $\{\phi_k\}$. The sparse coding is proposed by Olshausen & Field to explain the property of the simple cell in the brain[9]. Denoting part of input image patch pattern set as $\{I^p\}$, which have same size to the feature extraction vector ϕ_k , for training the feature extraction vectors where p is the pattern index. The idea of the sparse coding stands on the following points. One is the

image patch I^p should be expressed by a linear combination of the feature extraction vector $\{\phi_k\}$:

$$I^p \sim \sum_k a_k^p \phi_k. \quad (6)$$

And the other point is the almost all the coefficients a_k^p should be zero, that is only few feature extraction vectors support the image patch I^p , and we call under this condition as “sparse” state.

Then, we can introduce an objective function for the sparse coding as following:

$$J(\{\phi_k\}, \{a_k^p\}) = \sum_p \|I^p - \sum_k a_k^p \phi_k\|^2 + \lambda S(\{a_k^p\}), \quad (7)$$

$$S(\{a_k^p\}) = \sum_{p,k} \log(1 + (a_k^p)^2). \quad (8)$$

In the eq.(7), the first term means a data fitting term and the second means a constraint for sparseness, and the parameter λ controls the balance between these two terms. Minimizing the objective function for the $\{\phi_k\}$ and $\{a_k^p\}$, we can obtain the feature extracting vector set $\{\phi_k\}$ in the eq.(1).

3. Experiment

3.1 Materials

In order to evaluate our CAD system, we prepare 360 images, in which the number of each class are following: Consolidation:38, GGO:76, Honeycomb:49, Crazy-paving:37, Emphysema:54, Nodular:48, and Normal:58 cases. In usual, the HRCT image consists of 512×512 pixels. However, the whole image includes not only interest anatomy lung, but also another anatomies. Hence, in our system, we assume an input image is a part of HRCT image called “region of interest (ROI)”, which is segmented by a diagnostician. The size of ROI is configured as 32×32 pixels. Each ROI is segmented under the direction of a physician, and diagnosed by 3 physicians.

The acquisition parameters of those HRCT images are as follows: Toshiba “Aquilion 16” is used for imaging device, each slice image consists of 512×512 pixels, and pixel size corresponds to $0.546 \sim 0.826$ mm, slice thickness are 1 mm. The number of patients is 69 males and 42 females with age 66.3 ± 13.4 . The number of normal donor is 4 males and 2 females with age 44.3 ± 10.3 . The origin of these image data is provided Tokushima University Hospital.

On the consolidation image, we cannot recognize the vessels since lesion have too much high CT values such like water. GGO represents the light distributed lesion, and we can recognize vessels in contrast. Honeycomb appears geometrical patterns caused by the partial destruction of

alveoli. Crazy-paving represents mixture state GGO and honeycomb. These 4 cases are IIPs class. Emphysema represents distributed low CT values area caused by the destruction of alveoli. Nodular represents small (< 5 mm) nodule patterns. These 2 cases are not IIPs class, but another lung disease class. Normal class represents images collection from healthy donor.

3.2 Pre-processing for Input

Before carrying out the sparse coding, we adopt “sphering”, which is sometimes called pre-whitening, by principal component analysis (PCA). The purpose of the sphering is to normalize the signal represented by each pixel, and to eliminate the effect of cross correlation to other pixels. When we denote the $\{Y^p\}$ as the data set of raw pixel data of ROIs, the sphering process can be denoted as following:

$$\Lambda = \langle \mathbf{Y} \mathbf{Y}^T \rangle_p, \quad (9)$$

$$I^p = \Lambda^{-\frac{1}{2}} \mathbf{Y}^p, \quad (10)$$

where $\langle \cdot \rangle_p$ means the average over patterns indexed by p , and $\Lambda^{-\frac{1}{2}}$ can be obtained by eigenvalue decomposing using PCA. As the result of sphering, the cross-correlation matrix of pre-processed input, which denote as $\langle \mathbf{I} \mathbf{I}^T \rangle_p$, becomes a unit matrix, that is any pair of I_p have no cross correlation.

3.3 Evaluation method

In order to evaluate the ability of our CAD system, we apply leave one out cross-validation (LOOCV) method[10][11]. Applying this method, we left an input pattern for evaluation, and use another patterns to train the CAD system. Alternating the evaluation pattern, we evaluate the CAD system classification result on each occasion.

We fixed the number of feature vectors $\{\phi_k\}$ as 450 to satisfy overcomplete condition, and evaluated the effect of the vector length as $\{12 \times 12, 16 \times 16, 20 \times 20\}$. The balance parameter λ in eq.(7) is set as 1.0 that is decided experimentally. For Minimization of the cost function (7), we apply a method proposed by Olshausen & Field, that is a kind of gradient decent along the parameters $\{a_k^p\}$ and $\{\phi_k\}$ alternately[5]. Following equations are update rules:

$$\phi_k^{\text{new}} \leftarrow \phi_k + \eta \frac{\partial J}{\partial \phi_k} \quad (11)$$

$$a_k^p{}^{\text{new}} \leftarrow a_k^p + \eta \frac{\partial J}{\partial a_k^p} \quad (12)$$

where η is learning rate, that is fixed 0.0001 in this study. Eqs.(11) and (12) are applied alternately in the simulation.

Table 1: Classification ability by SCN with 20×20 feature extraction: Total correct ratio is 78.6%

	Classification result with SCN							ratio
	Cons.	GGO	Honey.	Crazy.	Emphy.	Nodul.	Norm.	
Consolidation	36	2	0	0	0	0	0	94.7%
GGO	0	59	2	2	0	13	0	77.6%
Honeycomb	0	3	44	2	0	0	0	89.8%
Crazy-Paving	0	3	0	28	1	5	0	75.7%
Emphysema	0	3	0	0	44	7	0	81.5%
Nodular	0	9	0	0	20	15	4	31.3%
Normal	0	0	0	0	1	0	57	98.3%

Table 2: Classification ability by GCN method with feature vector size 12×12 : Total correct ratio is 58.1%

	Classification result with GCN							ratio
	Cons.	GGO	Honey.	Crazy.	Emphy.	Nodul.	Norm.	
Consolidation	36	2	0	0	0	0	0	94.7%
GGO	0	57	5	3	6	1	4	75.0%
Honeycomb	1	5	36	3	1	3	0	73.5%
Crazy-Paving	1	13	7	16	0	0	0	43.2%
Emphysema	0	22	2	0	12	4	14	22.2%
Nodular	0	13	6	4	11	10	4	20.8%
Normal	0	4	0	0	10	2	42	72.4%

Table 3: Classification ability by SCN method with feature vector size 12×12 : Total correct ratio is 74.2%

	Classification result with SCN							ratio
	Cons.	GGO	Honey.	Crazy.	Emphy.	Nodul.	Norm.	
Consolidation	37	1	0	0	0	0	0	97.4%
GGO	0	56	1	6	6	7	0	73.7%
Honeycomb	0	2	44	3	0	0	0	89.8%
Crazy-Paving	0	10	10	16	0	0	1	43.2%
Emphysema	0	2	0	0	40	6	6	74.1%
Nodular	0	14	0	0	14	19	1	39.6%
Normal	0	0	0	0	3	0	55	94.8%

After training the feature extract vector set $\{\phi_k\}$, we can apply convolutional-net calculation shown in eqs.(3) and (5) where threshold parameter $\theta_k = 0.0$ for any k . As the result of convolutional-net calculation, we obtain a vector description, whose element is composed by $u_c(k)$, for each pattern I^p . Hence, we classify the vector to the IIPs' category, and we use the SVM as the classifier that is provided by OpenCV with default parameters[12].

Moreover, in order to compare the ability of our CAD with the conventional convolutional-net, we prepare Gabor function based system, that is $\{\phi_k\}$ as Gabor based system.In

the following, we abbreviate sparse coding convolutional-net as SCN, and Gabor filter base convolutional-net as GCN.

4. Results

Figure 3 shows the several examples of feature extract vectors of ϕ_k . Since the HRCT ROI images are not sort of natural images, the obtained bases ϕ_k are not similar to the Gabor filters that can be obtained by the sparse coding with natural scene processing[5]. This difference makes classification performance as following.

Table 1 shows the detail classification result by a confusion matrix. The Table 1 is a result of a SCN in which the length of feature vector ϕ_k is 20×20 network. This is the best result in our evaluation. Each row shows the input class, and each column shows the classification class. Thus diagonal line shown in bold numbers represents the number of correct classifications. For example, in the consolidation patterns, 37 cases are classified as consolidation correctly, 1 case is classified as GGO. The total correct ratio is shown in the last column. From the Table 1, we can see the correction ratio of all the classes except nodular class are over 75%. Especially, seeing the normal class column of the Table 1, a type II error called false negative, that is the failure probability of finding diseases, is nothing except nodular class. The nodular class is not category of IIPs, and its HRCT image does not have specific texture feature, but have only local sphere like patterns. Hence, the whitening pre-process, which is for normalization and elimination of cross correlation, may reduce this local feature, so that whitening may makes low classification ratio as the result. Anyway, improving of nodular class performance is a future work.

Table2 shows the result of classification performance by the GCN which is a modified model proposed by Kuwahara *et al.*[13]. Kuwahara *et al.* have applied Gabor filter for feature extraction, and AdaBoost for classification. We substitute this AdaBoost part for a SVM in order to compare with SCN. The scale of feature extractor ϕ_k is 12×12 that is the best one in the examined Gabor feature scales. Table 3 shows the GCN result of the same feature extractor scale. Comparing the Table 2 with the Table3, we can see the

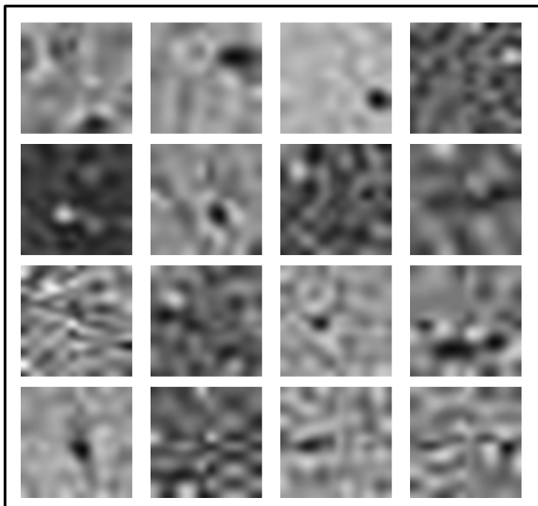


Figure 3: Several examples of feature extract vector ϕ_k obtained by sparse coding.

classification performance of the GCN have similar tendency to the SCN, however, total performance of the SCN is clearly improved from the GCN. Especially, we can see the performance for the emphysema and the crazy-paving classes are dominantly improved. Roughly speaking, the crazy-paving class is a intermediate image between GGO and Honeycomb, and we can estimate that Gabor based filters, which is used in the GCN for line or edge component extraction, are not sufficient for feature extraction.

Comparing Tables 1 and 2, which are different scale of ϕ_k , the performance of the large size ϕ_k is improved for the crazy-paving class. This result comes from the reducing of the miss classification to the honeycomb class, so that we can estimate large size ϕ_k is suitable for the extracting honeycomb structure.

5. Conclusion

In this study, we evaluated the sparse coding base convolutional-net for the multi-class IIP classification. Comparing the correction performance with the simple GCN that is a modified of the previous model, we can obtain an improvement result. Especially, type II error frequency of GCN is larger than that of the SCN. From the clinical point of view, we can conclude the several training method for the feature-extracting vector set $\{\phi_k\}$ is effective. Gangeh *et al.* also pointed out the similar tendency in their “texton” based model[6]. We consider the total performance of the classification rate is not so much bad, however, we should improve the performance of our SCN for the practical CAD system.

In the future works, in order to improve our SCN performance, we should find a tuning method or principle. In this work, we show the preliminary result for the feature extractor size effect. We can estimate the larger one is suitable for finding the structure such like crazy-paving and honeycomb, so that we should find optimal size of the feature extractor ϕ_k . One solution is that multi-scale feature extractor such like Lowe model may be effective for this problem[14].

Acknowledgement

We thank Professor Junji Ueno, Tokushima University. He provided us several advices for this study as well as a set of high resolution CT image of IIPs. This work is supported by Grant-in-Aids for Scientific Research (C) 21500214, and Innovative Areas 21103008, MEXT, Japan.

References

- [1] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.
- [2] H. Shouno, "Recent studies around the neocognitron," in *Neural Information Processing, 14th International Conference, ICONIP 2007, Kitakyushu, Japan, November 13-16, 2007, Revised Selected Papers, Part I*, ser. Lecture Notes in Computer Science, M. Ishikawa, K. Doya, H. Miyamoto, and T. Yamakawa, Eds., vol. 4984. Springer, 2007, pp. 1061–1070.
- [3] F. J. Huang and Y. LeCun, "Large-scale learning with svm and convolutional netw for generic object recognition." in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society CVPR'06, 2006.
- [4] M. Riesenhuber and T. Poggio, "Hierarchical model of object recognition in cortex," *Nature Neuroscience*, vol. 2, pp. 1019–1025, Nov. 1999.
- [5] B. A. Olshausen and D. J. Field, "Sparse coding with an overcomplete basis set: A strategy employed by v1?" *Vision Research*, vol. 37, no. 23, pp. 3311–3325, Dec. 1997.
- [6] M. J. Gangeh, L. Sorensen, S. B. Shaker, M. S. Kamel, M. de Bruijne, and M. Loog, "A texture-based approach for the classification of lung parenchyma in ct images," in *MICCAI*, ser. LNCS 6363, no. 3. Springer-Verlag Berlin Heidelberg, 2010, pp. 595–602.
- [7] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *J. Physiol.*, vol. 195, no. 1, pp. 215–243, 1968.
- [8] B. Shölkopf, K.-K. Sung, C. Burges, F. Girosi, P. Niyogi, T. Poggio, and V. Vapnik, "Comparing support vector machines with gaussian kernels to radial basis function classifiers," *IEEE Trans. on Signal Processing*, vol. 45, no. 11, pp. 2758–2765, Nov. 1997.
- [9] B. A. Olshausen and D. J. Field, "Emergence of simple-cell receptive field properties by learning a sparse code for natural images." *Nature*, vol. 381, pp. 607–609, Jun. 1996. [Online]. Available: <http://www.nature.com/nature/journal/v381/n6583/abs/381607a0.html>
- [10] M. Stone, "Cross-validation: A review." *Math. Operations. Stat. Ser. Stat.*, vol. 9, no. 1, pp. 127–139, 1978.
- [11] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2006.
- [12] G. Bradski, "The OpenCV Library," *Dr. Dobbs' Journal of Software Tools*, 2000.
- [13] M. Kuwahara, S. Kido, and H. Shouno, "Classification of patterns for diffuse lung diseases in thoracic ct images by adaboost algorithm," in *Proceedings of SPIE*, vol. 7260, February 2009.
- [14] J. Mutch and D. Lowe, "Object class recognition and localization using sparse features with limited receptive fields," *International Journal of Computer Vision*, vol. 80, no. 1, pp. 45–57, 2008. [Online]. Available: <http://www.springerlink.com/content/a608575053237603/fulltext.pdf>

Efficient and Approximate Simulation Algorithm of Kinetic Folding of an RNA Molecule

Takumi Tanigawa and Satoshi Kobayashi

Department of Computer Science
University of Electro-Communications
1-5-1, Chofugaoka, Chofu, Tokyo 182-8585, Japan

Abstract—Recently it is recognized as a very important research topic to simulate kinetic folding of an RNA molecule in order to understand its functionality *in vivo*. In this paper, we will propose a new approach to simulating kinetic folding of an RNA molecule based on a new idea of “enumerating secondary structures by a graph.” Although most of the previous works try to reduce the conformation space of a given RNA molecule in order to escape from the combinatorial explosion problem, the present paper gives us an efficient and approximate simulation methodology for hairpin formation with keeping the conformation space completely. As far as the authors’ knowledge, this is the first polynomial update time simulation algorithm for kinetic folding analysis of an RNA molecule which has a nice theoretical property that the convergence point of its simulation always exactly coincides with the equilibrium distribution of secondary structures of the RNA molecule. We evaluated the time efficiency and the accuracy of the proposed method against the exhaustive method which numerically simulates the master equation by completely generating all secondary structures. The results show that the proposed method is much faster than the exhaustive method and that the proposed method gives us well approximated simulation results.

Keywords: Kinetic Folding, Simulation, RNA, Equilibrium Computation

1. Introduction

RNA secondary structure plays important role in the biological function of many RNAs. Thus, the prediction of RNA structure is an important research topic in bioinformatics. One of the most effective method for such prediction is to use dynamic programming (DP) to obtain a minimum free energy (MFE, for short) structure ([15] [28] [21]). DP method is extendedly applied also to the calculation of equilibrium structure ensembles of RNA secondary structures([11]). These algorithms, however, can deal with only thermodynamical equilibrium, and not with kinetic effects on secondary structures (for instance, during the synthesis of RNA molecules). Furthermore, although stacking free energy of 5 base pairs is around 10 kcal/mol at 300 K, thermal energy kT is only 0.6 kcal/mol at 300 K, which implies that a native RNA may easily be trapped

into a suboptimal structure. Thus, the analysis of kinetic folding process of RNA molecules is very important for understanding their biological functions([13]).

A kinetical approach to RNA secondary structure prediction was introduced by Martinez ([10]), where folding kinetics is modeled by a Monte Carlo construction of secondary structures based on rate constants for iterative addition of complete helical regions, called helices, to some already existing structure. Modeling structure change by addition or deletion of helices is effective in reducing the conformation space of the RNA, thus, there are many works on RNA folding kinetics based on this formulation ([12], [1], [5], [13], [4], [7], [25]). However, the physical relevance of such moves seems debatable, because they cause large structural change per time step([3]). Furthermore, for a longer RNA sequence, we can not escape from the combinatorial explosion of conformation space even if we use helix based formulation.

Schmitz and Steger proposed a simulation method for kinetic folding of RNA secondary structures by using a Monte Carlo method based on rate constants for adding or removing a single base pair to some already existing structure([16]). The proposed move set is much more accurate than the helix based move set and was supported by many researchers ([3], [26], [18], [23], [14]), but, the combinatorial explosion problem of conformation space is more severe than the helix based approach.

In this paper, we will give a novel approach to simulating kinetic folding of an RNA molecule based on an elegant new idea of “enumerating secondary structures by a graph.” Although most of the previous works try to reduce the conformation space of a given RNA molecule in order to escape from the combinatorial explosion problem, the present paper will provide us with an efficient and approximate simulation methodology for hairpin formation with *keeping the conformation space completely*.

As far as the authors’ knowledge, this is the first polynomial update time approximate simulation algorithm for kinetic folding of an RNA molecule which has a nice theoretical property that the convergence point of its simulation always *exactly coincides with the equilibrium distribution* of secondary structures of the RNA molecule. Although we focus on secondary structures which are pseudoknot-free and

where k_{cal} is a calibration constant for adjusting simulation results with experimental results. We use the value $k_{cal} = 3.34 \times 10^6$, which was used in [16].

Most of the previous works have tried to reduce the conformation space in order to escape from its combinatorial explosion problem. In this work, however, we *completely* keep the space $C(X)$ and try to numerically simulate the master equation *efficiently and approximately* with the theoretical guarantee that the results will always reach to the *exact equilibria*. In order to achieve this goal, we will apply our previous theoretical work on the equilibrium analysis of chemical reaction systems in which molecules are interacting in various ways to generate tremendously many structures [8], [9]. Although the present paper deals with unimolecular reaction, the theory applies since the unimolecular reaction can be treated as a special case of the framework. In the next section, we will review the theory specialized to unimolecular reaction systems.

3. Enumeration Approach to Equilibria Analysis

Let X be a molecule and $C(X)$ be the conformation space, i.e., the set of structures, of X . Free energy of a conformation S of X is given by $F(S)$.

Assume that we have a *directed* graph $G = (V, Eg)$ with a finite set V of vertices and a finite set Eg of *directed* edges. For a vertex $v \in V$, by v_{in} and v_{out} , we denote the set of edges coming into v and going out from v , respectively. A vertex v with $v_{in} = \emptyset$ (with $v_{out} = \emptyset$, respectively) is called an *initial vertex* (a *final vertex*, respectively). By V_0 and V_f , we denote the set of initial and final vertices of G , respectively. A *simple* path of G is a path with each vertex appearing at most once. By $PT(G)$, we denote the set of simple paths starting from some vertex in V_0 and reaching to some vertex in V_f .

The essential part of the theory depends on the existence of a special *one-to-one* mapping ψ from $PT(G)$ to $C(X)$ satisfying the conditions explained below. After constructing such a mapping, the theory reduces the problem of computing equilibrium state to a convex optimization problem with respect to a set unknown variables whose size is $|Eg|$. Note that the cardinality of $PT(G)$ could be exponential with respect to $|Eg|$. Thus, the theory enables us to escape from the combinatorial explosion problem of the conformation space $C(X)$.

The requirement for the mapping ψ is *very simple* as follows. We ask the existence of a weight function ϵ on the edge set Eg such that for every path $\gamma \in PT(G)$, $F(\psi(\gamma)) = \sum_{e \in Eg \text{ s.t. } e \in \gamma} \epsilon(e)$ holds. This condition means that for every $\gamma \in PT(G)$, the sum of weight of edges appearing in γ equals to the free energy of the corresponding structure $\psi(\gamma)$ of the path γ . Intuitively speaking, every edge in the graph G corresponds to some local structure

of conformation space, and its weight is just the free energy of the corresponding local structure. In case of equilibrium analysis of an RNA molecule at the secondary structure level, it would be expected that we can construct a graph whose edge would correspond to local structures, such as hairpin loops, bulge loops, internal loops, etc. An example of such enumeration graphs will be given in the next section 4.

4. Enumerating Secondary Structures of an RNA

We will give an example of graphs by which we can enumerate all linear secondary structures of an RNA sequence.

Let $X = x_1 \cdots x_n$ be an RNA sequence. Then, we prepare a set of vertices corresponding to base pairs which may form in the sequence X . Moreover, we use two additional special vertices: an initial vertex s and a final vertex f . The construction of edge set is as follows. We draw an edge from a base pair (i, j) to a base pair (k, l) if and only if $i < k < l < j$ holds. Furthermore, for every base pair bp , we put an edge from s to bp and an edge from bp to f . Formally, we can define a graph $G = (V, Eg)$ for the sequence X :

$$\begin{aligned} BP &= \{(i, j) \mid 1 \leq i < j \leq n, (x_i, x_j) \in WC\}, \\ V &= \{s, f\} \cup BP, \\ Eg &= \{(s, bp) \mid bp \in BP\} \cup \{(bp, f) \mid bp \in BP\} \cup \\ &\quad \{(i, j), (k, l) \mid (i, j), (k, l) \in BP, i < k < l < j\}. \end{aligned}$$

A path in $PT(G)$ for G defined above naturally corresponds to a linear secondary structure consisting of base pairs contained in it. An example of graphs for enumerating secondary structures of the sequence $X = \text{GGAAACUU}$ is given in Figure 3.

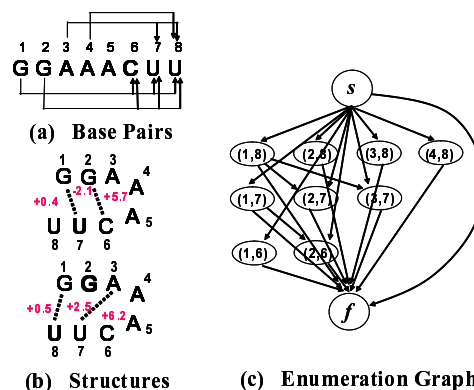


Fig. 3: An Example of Enumeration Graphs

Figure 3 (a) illustrates all possible base pairs of the sequence X . Figure 3 (c) shows an enumeration graph for the sequence X . A path $s \rightarrow (1, 7) \rightarrow (2, 6) \rightarrow f$ corresponds to the upper secondary structure in Figure 3 (b). A path $s \rightarrow$

$(1, 8) \rightarrow (3, 7) \rightarrow f$ corresponds to the lower secondary structure in Figure 3 (b). In this way, we can enumerate all linear secondary structures of X . The mapping from a path to its corresponding secondary structure is denoted by ψ .

As is clear from the above example, an edge between the base pairs (i, j) and (k, l) in the graph corresponds to a local loop structure (either of stacked base pairs, a bulge, or an internal loop) surrounded by (i, j) and (k, l) . An edge between s (f) and a base pair (i, j) corresponds to a free end loop outside (a hairpin loop closed by) the base pair (i, j) . Thus, the weight $\epsilon(e)$ of an edge e is defined as the free energy of the corresponding local secondary structure of e . For instance, the free energy values of local secondary structures are given as real values in Figure 3 (c). Thus, the weight of edges $s \rightarrow (1, 7)$, $(1, 7) \rightarrow (2, 6)$, $(2, 6) \rightarrow f$, $s \rightarrow (1, 8)$, $(1, 8) \rightarrow (3, 7)$, $(3, 7) \rightarrow f$ are given by $+0.4$, -2.1 , $+5.7$, $+0.5$, $+2.5$, $+6.2$, respectively.

5. Efficiently Computing Equilibria by Convex Programming

Let X be a molecule and $C(X)$ be a conformation space of X . An *equilibrium distribution* of $C(X)$ is a probability distribution $[]$ over $C(X)$ such that for any conformations S_1 and S_2 in $C(X)$, the following equilibrium equation holds:

$$\frac{[S_2]}{[S_1]} = e^{-\frac{F(S_2) - F(S_1)}{RT}}.$$

When we succeed in constructing an enumeration graph G for a conformation space $C(X)$ of a molecule X satisfying the conditions explained in section 3, following a general theory developed by the second author of this paper ([8], [9]), we can efficiently compute an equilibrium distribution by solving a minimization problem explained below.

First, we will introduce an unknown variable w_e for each edge e of the graph G . The variable w_e takes a real value between 0 and 1, and represents a probability of the local substructure corresponding to e existing in the current probability distribution over $C(X)$.

For convenience, for every $v \in V - V_0 - V_f$, we define $w_v = \sum_{e \in v_{out}} w_e$. Consider the following minimization problem:

Minimization Problem P1
minimize :

$$FE((w_e \mid e \in Eg)) \stackrel{def}{=} \sum_{e \in Eg} \frac{\epsilon(e)}{RT} \cdot w_e + \sum_{e \in Eg} w_e (\log w_e - 1) - \sum_{v \in V - V_0 - V_f} w_v (\log w_v - 1)$$

subject to :

$$\begin{aligned} \sum_{v \in V_0} \sum_{e \in v_{out}} w_e &= 1, \\ \sum_{e \in v_{in}} w_e &= \sum_{e \in v_{out}} w_e, \quad (\forall v \in V - V_0 - V_f) \\ w_e &\geq 0, \quad (\forall e \in Eg) \end{aligned}$$

where unknown variables are w_e 's ($e \in Eg$) and recall that w_v 's are sums of variables w_e 's.

Then, the following theorem was proved in [8]:

Theorem 1: Consider a minimizer $(w_e \mid e \in Eg)$ of the above minimization problem *P1*. Then, an equilibrium distribution is given by: for any $S \in C(X)$,

$$[S] = \frac{\prod_{e \in Eg \text{ s.t. } e \in \psi^{-1}(S)} w_e}{\prod_{v \in V - V_0 - V_f \text{ s.t. } v \in \psi^{-1}(S)} w_v}, \quad (2)$$

In order to obtain an equilibrium distribution of an RNA molecule at the secondary structure level, we should first obtain a minimizer of the optimization problem *P1* based on the graph G given in section 4. This is achieved efficiently since the objective function of the problem *P1* is convex as shown in [8], and thus, we can apply a convex programming method to obtain a minimizer. Equilibrium distribution is then obtained by the expression (2).

6. The Objective of This Work

In this way, we will be able to efficiently compute an equilibrium distribution of an RNA molecule. This is not, however, the main purpose of this paper. In this work, we aim at efficiently simulating kinetic folding process specified by the master equation (1). Thus, applying convex programming method might not lead us to the goal of this paper. We need to carefully choose a descending direction of the objective function of the optimization problem *P1*. Such a careful choice of the direction will be proposed in section 7, and the validity of the choice will be shown theoretically in section 8. This theoretical argument guarantees that the proposed simulation algorithm will always converge to an equilibrium distribution. Our method is distinguished from the others in the *convergence property to equilibria*.

7. Algorithm

In this section, we will give an algorithm for efficiently and approximately simulating the kinetic folding process of an RNA molecule at the secondary structure level. The algorithm is presented with intuitive explanation of the reason why we will obtain the algorithm. The key idea behind the algorithm is to *locally interpret in the graph representation the kinetic moves of Add and Delete*.

We first explain how to interpret the move *Add* in view of enumeration graph (Figure 4). The move *Add* inserts a

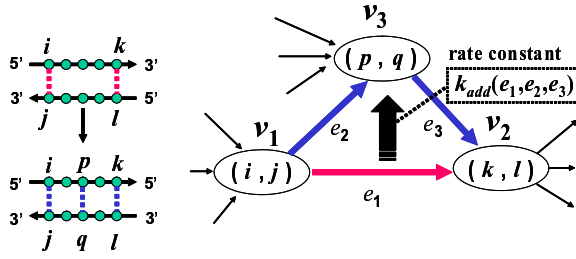


Fig. 4: Local Interpretation of Add Move

new base pair to the current conformation. Consider an *Add* move which inserts a base pair (p, q) between two base pairs (i, j) and (k, l) . Note that $i < p < k < l < q < j$ holds. This move can be interpreted in the enumeration graph representation as a move from a path containing the edge $(i, j) \rightarrow (k, l)$ to another path containing the edges $(i, j) \rightarrow (p, q)$ and $(p, q) \rightarrow (k, l)$ keeping the probabilities of the other edges unchanged (See Figure 4). Based on this observation, it is very natural to interpret the *Add* operation locally as the change of probabilities of the three edges, $(i, j) \rightarrow (k, l)$, $(i, j) \rightarrow (p, q)$, and $(p, q) \rightarrow (k, l)$, as follows:

$$\Delta w_{e_1} = -k_{add}(e_1, e_2, e_3) \cdot w_{e_1} \Delta t, \quad (3)$$

$$\Delta w_{e_2} = k_{add}(e_1, e_2, e_3) \cdot w_{e_1} \Delta t, \quad (4)$$

$$\Delta w_{e_3} = k_{add}(e_1, e_2, e_3) \cdot w_{e_1} \Delta t, \quad (5)$$

where $k_{add}(e_1, e_2, e_3)$ is a rate constant for this local reaction which causes the change of probabilities of the edges e_1, e_2 and e_3 , which is defined by:

$$k_{add}(e_1, e_2, e_3) = \begin{cases} e^{-\frac{\epsilon(e_2) + \epsilon(e_3) - \epsilon(e_1)}{RT}} & \text{if } \epsilon(e_2) + \epsilon(e_3) - \epsilon(e_1) \geq 0 \\ 1 & \text{otherwise} \end{cases}$$

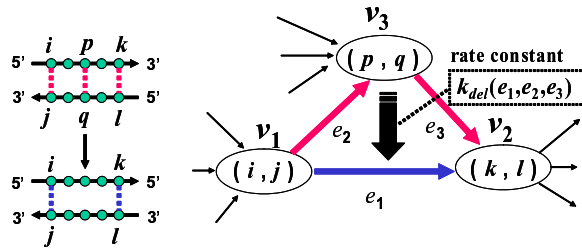


Fig. 5: Local Interpretation of Delete Move

Next we consider the case of the move *Delete*. The move *Delete* removes a base pair from the current conformation. Consider a *Delete* move which removes a base pair (p, q) from between two base pairs (i, j) and (k, l) . Note that $i < p < k < l < q < j$ holds also in this case. This move can be interpreted in the enumeration graph representation as a

move from a path containing the edges $(i, j) \rightarrow (p, q)$ and $(p, q) \rightarrow (k, l)$ to another path containing the edge $(i, j) \rightarrow (k, l)$ keeping the probabilities of the other edges unchanged (See Figure 5). Thus, it is natural to interpret the *Delete* operation locally as the change of probabilities of the three edges, $(i, j) \rightarrow (k, l)$, $(i, j) \rightarrow (p, q)$, and $(p, q) \rightarrow (k, l)$, as follows:

$$\Delta w_{e_1} = k_{del}(e_1, e_2, e_3) \cdot w(e_2, e_3) \Delta t, \quad (6)$$

$$\Delta w_{e_2} = -k_{del}(e_1, e_2, e_3) \cdot w(e_2, e_3) \Delta t, \quad (7)$$

$$\Delta w_{e_3} = -k_{del}(e_1, e_2, e_3) \cdot w(e_2, e_3) \Delta t, \quad (8)$$

where $w(e_2, e_3)$ is the probability of the paths passing through both of the edges e_2 and e_3 , and $k_{del}(e_1, e_2, e_3)$ is a rate constant for this local reaction which causes the change of probabilities of the edges e_1, e_2 and e_3 , which is defined by:

$$k_{del}(e_1, e_2, e_3) = \begin{cases} e^{-\frac{\epsilon(e_1) - \epsilon(e_2) - \epsilon(e_3)}{RT}} & \text{if } \epsilon(e_1) - \epsilon(e_2) - \epsilon(e_3) \geq 0 \\ 1 & \text{otherwise} \end{cases}$$

Note that we still have difficulty in this local interpretation of the move *Delete*, because we do not have any information about the probability $w(e_2, e_3)$. We only know about probabilities of edges e_2 and e_3 as w_{e_2} and w_{e_3} , respectively. So, we should approximately guess the probability $w(e_2, e_3)$ of paths passing through both of edges e_2 and e_3 . We will propose to use the following estimate:

$$w(e_2, e_3) = \begin{cases} \frac{w_{e_2} \cdot w_{e_3}}{w_{v_3}}, & \text{if } w_{v_3} > 0 \\ 0 & \text{otherwise} \end{cases}$$

where v_3 is the vertex corresponding to the base pair (p, q) . This expression intuitively means that every path coming to the vertex v_3 is splitted into all directions from v_3 proportionally to the probability distribution of edges going out from v_3 . This estimate theoretically guarantees that the proposed method will always reach to an equilibrium distribution at the convergence point, as will be shown in the next section 8.

The proposed method will apply the above rule of local probability change to every triple of edges located in a triangular form as illustrated in Figure 4 and Figure 5. It is clear that the time complexity of the update of probabilities of all w_e 's ($e \in Eg$) is bounded by a polynomial function with respect to the length of the sequence X .

8. Theoretical Analysis of the Algorithm

Theorem 2: The direction specified by the expressions (3)-(8) is a decending direction of the optimization problem *PI*. *Proof:* For every triangle consisting of edges e_1, e_2, e_3 and vertices v_1, v_2, v_3 as illustrated in Figure 4 and

5, we have:

$$\begin{aligned}\frac{\partial FE((w_e | e \in Eg))}{\partial w_{e_1}} &= \frac{\epsilon(e_1)}{RT} + \log w_{e_1} - \log w_{v_1}, \\ \frac{\partial FE((w_e | e \in Eg))}{\partial w_{e_2}} &= \frac{\epsilon(e_2)}{RT} + \log w_{e_2} - \log w_{v_1}, \\ \frac{\partial FE((w_e | e \in Eg))}{\partial w_{e_3}} &= \frac{\epsilon(e_3)}{RT} + \log w_{e_3} - \log w_{v_3},\end{aligned}$$

and the sum of the moves *Add* and *Delete*, denoted by (d_1, d_2, d_3) , is given by:

$$\begin{aligned}d_1 &= -k_{add}(e_1, e_2, e_3)w_{e_1} + k_{del}(e_1, e_2, e_3)\frac{w_{e_2}w_{e_3}}{w_{v_1}}, \\ d_2 &= k_{add}(e_1, e_2, e_3)w_{e_1} - k_{del}(e_1, e_2, e_3)\frac{w_{e_2}w_{e_3}}{w_{v_1}}, \\ d_3 &= k_{add}(e_1, e_2, e_3)w_{e_1} - k_{del}(e_1, e_2, e_3)\frac{w_{e_2}w_{e_3}}{w_{v_1}}.\end{aligned}$$

Then, we have:

$$\begin{aligned}\sum_{i=1}^3 \frac{\partial FE((w_e | e \in Eg))}{\partial w_{e_i}} \cdot d_i &= \\ &k_{add}(e_1, e_2, e_3)w_{e_1} \left(1 - \frac{k_{del}(e_1, e_2, e_3)w_{e_2}w_{e_3}}{k_{add}(e_1, e_2, e_3)w_{e_1}w_{v_3}}\right) \times \\ &\left(\log \frac{w_{e_2}w_{e_3}}{w_{e_1}w_{v_3}} e^{\frac{\epsilon(e_2)+\epsilon(e_3)-\epsilon(e_1)}{RT}}\right) = \\ &k_{add}(e_1, e_2, e_3)w_{e_1} \left(1 - \frac{w_{e_2}w_{e_3}}{w_{e_1}w_{v_3}} e^{\frac{\epsilon(e_2)+\epsilon(e_3)-\epsilon(e_1)}{RT}}\right) \times \\ &\left(\log \frac{w_{e_2}w_{e_3}}{w_{e_1}w_{v_3}} e^{\frac{\epsilon(e_2)+\epsilon(e_3)-\epsilon(e_1)}{RT}}\right) \leq 0,\end{aligned}$$

where we use:

$$\frac{k_{del}(e_1, e_2, e_3)}{k_{add}(e_1, e_2, e_3)} = e^{\frac{\epsilon(e_2)+\epsilon(e_3)-\epsilon(e_1)}{RT}},$$

completing the proof.

Since the objective function of *PI* is convex, by Theorem 2, we can conclude that the simulation by the proposed method will reach to an equilibrium distribution.

9. Simulation Results

We have done two kinds of computational experiments. The first one is for evaluating the time efficiency of the proposed method against the exhaustive method, in which for an input sequence X , we generated all the secondary structures in $C(X)$, and simulated the folding kinetics of X based on the master equation (1). Simulations of both methods start from a random chain structure.

The other experiment is for showing that the proposed method gives us a well approximated simulation result for structures which are dominant at equilibrium.

In this section, we will give some computational experimental results which will show that the proposed method is very time efficient compared to the exhaustive method and it gives us a fairly well approximated kinetic folding paths. The tested sequences are listed in Table 1.

No.	length	Sequence
1	10	AGCCGUUUCC
2	12	AACCCUACCCUU
3	14	GGGCGAAACGCCCU
4	16	GCCGCGAAACGCGGCC
5	18	CGGGCCGAAAUCCGCCCU
6	20	CGGGCGGAAAUCCGCCCU

Table 1: RNA Sequences

No.	N_{str}	T_E	T_P
1	15	0.27s	0.09s
2	14	0.27s	0.08s
3	200	7.34s	0.84s
4	322	12.99s	1.13s
5	832	38.57s	2.98s
6	3293	2m58.29s	9.40s

Table 2: Time Efficiency Result

For a given sequence X , we did kinetic simulations starting from a random chain structure by using the exhaustive method and the proposed method up to 1,000 time steps, where we use $\Delta t = 1.0 \times 10^{-8}$ sec. The time for executing 1,000 step simulation is given in Table 2, where T_E is for the exhaustive method and T_P for the proposed method. The number of structures in $C(X)$ is given in the column N_{str} .

In Fig.6 and Fig.7, we simulated dominant structures of the sequence ACGUGCACAAAAGUGCACGU of length 20. The optimal structure is (((((((((.....)))))))) (-12.0 kcal/mol) and its simulation result is shown in Fig.6. Suboptimal structures are St1= (((((((.....)))))) (-10.0 kcal/mol) and St2= ..((((((.....)))))).. (-9.9 kcal/mol), and their simulation results are shown in Fig.7. In both of the figures, the lines specified by ‘‘E’’ and ‘‘P’’ represent the simulation results by the *Exhaustive* and the *Proposed* methods, respectively. Simulations by the proposed method give us well approximated results compared to the exhaustive (i.e., exact) simulations.

Concerning rare structures, the time step Δt should be carefully chosen as small values enough, since concentrations of rare structures are very sensitive to large Δt , which result in incorrect simulations in both of the exhaustive and the proposed methods. The topic on the choice of appropriate Δt would be a future research topic.

10. Conclusion

We proposed a novel method for efficiently and approximately numerically simulating kinetic folding process of an RNA molecule based on the idea of ‘‘enumerating conformations by a graph.’’ The proposed method has a very nice theoretical property that the convergence point of simulation results exactly coincides with the equilibrium. Time efficiency, the accuracy and the effectiveness of the method were shown by computational experiments.

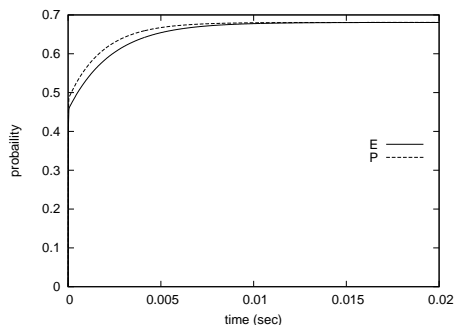


Fig. 6: Simulations of Optimal Structure

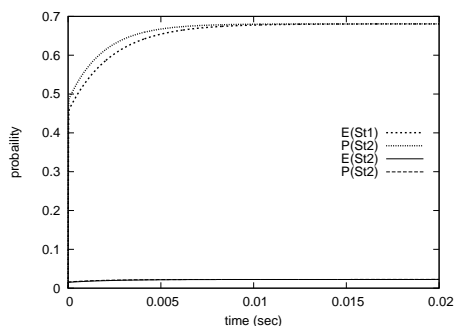


Fig. 7: Simulations of Suboptimal Structures

The current implementation is restricted only to the class of linear secondary structures, i.e. structures which do not contain branches. But, the proposed method can be extended to a more broader class of secondary structures if we prepare an appropriate enumeration graph for the extended structure class. Thus, it is an important future research topic to find such an enumeration scheme for a broader class of secondary structures.

In this paper, we evaluated the accuracy of the proposed simulation method only by computational experiments. Theoretical analysis of the accuracy of the method is also an important open problem. Furthermore, based on this kind of theoretical analysis, it might be interesting to improve the method in order to achieve a better accuracy.

Acknowledgement

The second author is supported in part by Grant-in-Aid for Scientific Research (C) No.22500010, Japan Society for the Promotion of Science.

References

- [1] Abrahams, J.P., van den Berg, M., van Batenburg, E., and Pleij, C. (1990), Prediction of RNA secondary structure, including pseudoknotting, by computer simulation, *Nucleic Acids Research*, **18**, 3035-3044.
- [2] Bartley, L.E., Zhuang, X., Das, R., Chu, S., and Herschlag, D. (2003), Exploration of the transition state for tertiary structure formation between an RNA helix and a large structure RNA, *J. Mol. Biol.*, **328**, 1011-1026.
- [3] Flamm, C., Fontana, W., and Hofacker, I.L. (2000), RNA folding at elementary step resolution, *RNA*, **6**, 325-338.
- [4] Galzitskaya, O.V., and Finkelstein, A.V. (1996), Computer simulation of secondary structure folding of random and "edited" RNA chains, *J. Chem. Phys.*, **105**, 319-325.
- [5] Gulyaev, A.P., van Batenburg, F.H.D., and Pleij, C. (1995), The Computer Simulation of RNA Folding Pathways Using a Genetic Algorithm, *J. Mol. Biol.*, **250**, 37-51.
- [6] Hofacker, I.L., Fontana, W., Stadler, P.F., Bonhoeffer, L.S., Tacker, M., Schuster, P. (1994), Fast folding and comparison of RNA secondary structures (the Vienna RNA package), *Monatshefte für Chemie*, **125**, 167-188.
- [7] Isambert, H., and Siggia, E.D. (2000), Modeling RNA folding paths with pseudoknots: Application to hepatitis delta virus ribozyme, *PNAS*, **97**, 6515-6520.
- [8] Kobayashi, S. (2007), A new approach to computing equilibrium state of combinatorial hybridization reaction systems, in *Proc. of Computing and Communications from Biological Systems: Theory and Applications*, Budapest, Hungary, CD-ROM, paper2376. (Extended full version is available at http://comp.cs.uec.ac.jp/~satoshi/TR_CS0801rev.pdf)
- [9] Kobayashi, S. (2008), A software tool for analyzing combinatorial hybridization reaction systems, in *Proc. of 14th International Meeting on DNA Based Computer*, Track B, oral presentation.
- [10] Martinez, H.M. (1984), An RNA folding rule, *Nucleic Acids Research*, **12**, 323-334.
- [11] McCaskill, J.S. (1990), The equilibrium partition function and base pair binding probabilities for RNA secondary structure, *Biopolymers*, **29**, 1105-1119.
- [12] Mirnov, A.A., Dyakonova, L.P., and Kister, A.E. (1985), A kinetic approach to the prediction of RNA secondary structures, *Journal of Biomolecular Structure and Dynamics*, **2**, 953-962.
- [13] Morgan, S.R., and Higgs, P.G. (1996), Evidence for kinetic effects in the folding of large RNA molecules, *J. Chem. Phys.*, **105**, 7152-7157.
- [14] Ndifon, W. (2005), A complex adaptive systems approach to the kinetic folding of RNA, *BioSystems*, **82**, 257-265.
- [15] Nussinov, R., Pieczenik, G., Griggs, J.R., and Kleitman, D.J. (1978), Algorithms for Loop Matchings, *SIAM J. Appl. Math.*, **35**, 68-82.
- [16] Schmitz, M., and Steger, G. (1996), Description of RNA folding by "Simulated Annealing", *J. Mol. Biol.*, **255**, 254-266.
- [17] Sosnick, T.R., and Pan, T. (2003), RNA folding: models and perspectives, *Curr. Opin. Struct. Biol.*, **13**, 309-316.
- [18] Tang, X., Kirkpatrick, B., Thomas, S., Song, G., and Amato, N.M. (2004), Using motion planning to study RNA folding kinetics, in *Proc. of 8th Annual International Conference on Research in Computational Molecular Biology (RECOMB'04)*, 252-261.
- [19] Thirumalai, D., Lee, N., Woodson, S.A., Klimov, D.K. (2001), Early events in RNA folding, *Annu. Rev. Phys. Chem.*, **52**, 751-762.
- [20] Treiber, D.K., and Williamson, J.R. (2001), Beyond kinetic traps in RNA folding, *Curr. Opin. Struct. Biol.*, **11**, 309-314.
- [21] Uemura, Y., Hasegawa, A., Kobayashi, S., and Yokomori, T. (1999), Tree Adjoining Grammars for RNA Structure Prediction, *Theoretical Computer Science*, **210**, 277-303.
- [22] Uhlenbeck, O.C. (1990), Nucleic-acid structure — tetraloops and RNA folding, *Nature*, **346**, 613-614.
- [23] Wolfinger, M.T., Svrek-Seiler, W.A., Flamm, C., Hofacker, I.L., and Stadler, P.F. (2004), Efficient computation of RNA folding dynamics, *J. Phys. A: Math. Gen.*, **37**, 4731-4741.
- [24] Woodson, S.A. (2000), Recent insights on RNA folding mechanisms from catalytic RNA, *Cell. Mol. Life Sci.*, **57**, 796-808.
- [25] Xayaphoummine, A., Bucher, T., and Isambert, H. (2005), Kinofold web server for RNA/DNA folding path and structure prediction including pseudoknots and knots, *Nucleic Acids Research*, **33**, Web Serve issue, W605-W610.
- [26] Zhang, W., and Chen, S.-J. (2002), RNA hairpin-folding kinetics, *PNAS*, **99**, 1931-1936.
- [27] Zhang, W., and Chen, S.-J. (2006), Exploring the complex folding kinetics of RNA hairpins: I: General folding kinetics analysis, *Biophysical Journal*, **90**, 765-777.
- [28] Zuker, M., and Steigler, P. (1981), Optimal Computer Folding of Large RNA Sequences using Thermodynamics and Auxiliary Information, *Nucleic Acids Research*, **9**, 133-148.

DNA Logic Circuits with a DNA Polymerase and a Nicking Enzyme

Ryo Hirose¹, Satoshi Kobayashi¹, Ken Komiya²

¹Department of Communication Engineering and Informatics
Graduate School of Informatics and Engineering
University of Electro-Communications, Tokyo, Japan

²Department of Computational Intelligence and Systems Science
Interdisciplinary Graduate School of Science and Engineering
Tokyo Institute of Technology, Yokohama, Japan

Abstract—*The current most popular and established approach to DNA logic circuits is the implementation by DNA strands reaction networks where toehold-mediated strand displacement invokes and performs the evaluation of each DNA logic gate. Strand displacement approach requires, however, a large amount of time, for instance, approximately 30 minutes, to execute a logic operation ([18]). Furthermore, the concentration of an output molecule released from a gate can not exceed that of the gate molecule. Therefore, it is often the case that large quantities of input and gate molecules are required when the gate is of large out-degree. In order to overcome these problems, it is indispensable to devise a DNA logic gate which runs quickly and can amplify the quantity of the output molecule. We will propose a DNA implementation of logic gates with such good properties using DNA polymerase and nicking enzyme.*

Keywords: DNA Computing, Logic Circuit, DNA Polymerase, Nicking Enzyme

1. Introduction

In 1994, Adleman initiated the paradigm of DNA computing by devising and demonstrating a biological experimental protocol to solve Directed Hamiltonian Path Problem([1]). It has achieved important progress of the technologies in making DNAs work as computing devices([4][3][2], etc.). They also produce some new and important key technologies([14][10], etc.) in the field of DNA nanotechnology, where it is aimed to construct intended nano-scale shapes or structures by self-assembly of DNA molecules. In these fields, it is currently emerging the movement for establishing the methodology to construct automatic molecular robots performing intended tasks in some specified environment ([8][5], etc.). A molecular robot should, however, contain at least three important components, a sensor, a circuit, and an actuator, in itself. The current technology ([8][5]) does not satisfy these three requirements, and it is still at a premature stage. So, it is very challenging to explore a possible framework of the methodology to construct a molecular robot with a sensor, a circuit, and an actuator.

Molecular circuits equipped in molecular robots should serve as logic circuits, memory devices, and control devices, etc ([12]). In this paper, we will focus on a molecular circuit as a computing device performing logic operations. There are many works which have proposed DNA implementations of logic circuits. One of the most popular and established approach to DNA logic circuits is the implementation by DNA strands reaction networks where toehold-mediated strand displacement ([16]), strand displacement for short, invokes and performs the evaluation of each DNA logic gate([17][15][18]). Strand displacement approach requires, however, a large amount of time, for instance, approximately 30 minutes, to execute a logic operation([18]). Furthermore, the concentration of an output molecule released from a gate can not exceed that of the gate molecule. Therefore, it is often the case that large quantities of input and gate molecules are required when the gate is of large out-degree. In order to overcome these problems, it is indispensable to devise a DNA logic gate which runs quickly and can amplify the quantity of the output molecule. We will propose a DNA implementation of logic gates with such good properties using a DNA polymerase and a nicking enzyme.

2. Preliminaries

DNA is a molecule consisting of simple units, called *nucleotides*, with backbones made of sugars and phosphate groups joined by phosphodiester bonds. Each nucleotide contains one of four types of atomic groups, called bases, each being either of adenine (abbreviated A), cytosine (C), guanine (G) and thymine (T). These long polymers are more commonly called *strands*, and short polymers are called *oligonucleotides*, or simply *oligos*. Note that every DNA strand has two distinct ends, one with a free 5' phosphate group and the other with a free 3' hydroxyl group, referred to as the 5' and 3' ends, respectively. Thus, we can regard a sequence of nucleotides as having a natural orientation from its 5' to 3' ends, and it is often written as a sequence of letters (bases) A, C, G, T oriented from 5' to 3' direction.

Under some appropriate chemical conditions, two strands will pair up and twist around each other to form a fa-

mous double helix structure discovered by Watson and Crick. The pairing happens between A's and T's and between G's and C's, which is called Watson-Crick base pairing. This pairing occur only if the two strands runs in an antiparallel fashion. For instance, the strands $5'$ -GCATCAG- $3'$ and $5'$ -CTGATGC- $3'$ will form a hybridized object $\begin{matrix} 5' - \text{GCATCAG} - 3' \\ 3' - \text{CGTAGTC} - 5' \end{matrix}$. These kinds of completely hybridized DNA strands are called *double strands*. A strand with no base pairing with other strands is called a *single strand*.

For any DNA single strand X (where X is a sequence of bases), by X^* we denote the complementary DNA strand of X , i.e. the sequence obtained from X by replacing each A by T, each T by A, each C by G, and each G by C, and reversing the order. For instance, for $X = 5'$ -GCATCAG- $3'$, we have $X^* = 5'$ -CTGATGC- $3'$. For any single strand X , X and X^* will form a double strand, which we will denote by $\begin{matrix} 5' - X - 3' \\ 3' - X^* - 5' \end{matrix}$.

DNA polymerase is an enzyme, or molecular machine, which reads a single strand X in the $3'$ to $5'$ direction, and builds the complementary strand X^* in the $5'$ to $3'$ direction, one nucleotide at a time, where the strand X is called a template. The activation of DNA polymerase to work as a (complementary) copy machine requires a short portion of double stranded part in the template. That is, we need a short piece of single strand, called *primer*, which is complementary to some part of the template. It is onto the $3'$ end of this primer that DNA polymerase will add new nucleotides. So, in order to make the copy X^* of X , we need a primer which is complementary to the last $3'$ end portion of the strand X .

Restriction endonucleases recognize specific nucleotide sequences in the double-stranded form and generally cleave both strands. Some sequence-specific endonucleases, however, cleave only one of the strands. These endonucleases are called nicking endonucleases, or nicking enzymes. For instance, a nicking enzyme Nt.BsmAI recognizes the double stranded sequence $\begin{matrix} 5' - \text{GTCTCN} / \text{N} - 3' \\ 3' - \text{CAGAGN} \text{N} - 5' \end{matrix}$, and cleaves the upper strand at the position indicated by the symbol $/$, where N can be either of the four bases.

3. Related Work

The current most promising approach to the construction of DNA logic gates is the DNA implementation of logic operation by the use of strand displacement. Its basic mechanism is illustrated in Figure 1. Consider a complex consisting of strands $5' - A - B - 3'$ and $5' - C^* - B^* - 3'$ with B and B^* hybridized, where A, B, C are sequences of bases. If another strand $5' - B - C - 3'$ exists in a solution, then $5' - B - C - 3'$ hybridizes to the complex with C and C^* hybridized. The strand $5' - B - C - 3'$ gradually extends its pairing with $5' - C^* - B^* - 3'$ by replacing the strand

$5' - A - B - 3'$ in a random walk fashion. This process is called "branch migration".

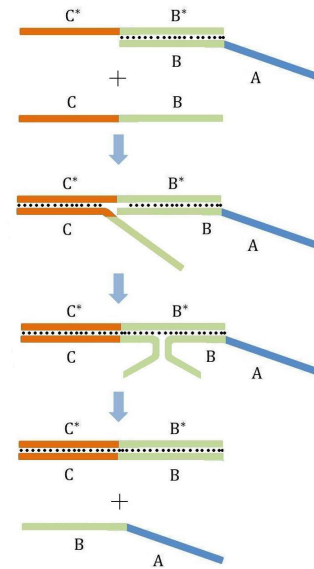


Fig. 1: Toehold-mediated Strand Displacement

There have been many research works to propose bio-lab methods to implement some computational tasks by the use of strand displacement ([11][17][15][18],etc.), and it is considered as one of the most promising bio-lab techniques in DNA computing. The computational capability of strand displacement is also an interesting research topic from the viewpoint of computation theory([12][9],etc.).

Strand displacement is, however, a slow reaction if we want to use it as an essential bio-lab operation to implement DNA logic circuits of molecular robots. Furthermore, in these approaches, the concentration of an output molecule released from a gate can not exceed that of the gate molecule. In this sense, the construction of large circuits based only on strand displacement is not feasible.

In the next section, we will propose a new bio-lab method for constructing logic circuits with DNA molecules, where we use the amplification system based on nicking enzyme and DNA polymerase. Waker first applied this reaction to the construction of isothermal amplification system of DNA ([13]). The idea of using nicking enzyme to computational molecular devices was also accomplished by Matsuda and Yamamura ([6]), where they used a DNA polymerase and a nicking enzyme in order to cascade molecular state transition systems based on Whiplash PCR([3]). Very recently, Montagne, et al., proposed to use a DNA polymerase and a nicking enzyme to construct a programmed DNA oscillator([7]). In this paper, we will apply a similar reaction system to the construction of DNA logic circuits, and we will show by computational simulations that the proposed system is more efficient than the strand displacement approaches.

Furthermore, in the proposed system, the output of each DNA logic gate can be amplified by the use of a DNA polymerase and a nicking enzyme.

4. Construction of Combinatorial Circuit

In this section, we will propose a method to construct combinatorial circuits by using DNA polymerase and nicking enzyme. As in other related works about DNA logic gates, we will prepare for each boolean variable x , a DNA strand X . The existence of X in the solution implies the evaluation of the boolean variable x as 1. On the other hand, how can we encode the evaluation $x = 0$ into the solution? For this purpose, we will also prepare another DNA strand NX , and we regard the existence of NX in the solution as the evaluation of the variable x as 0. Thus, the strand X and NX can not exist at the same time in the solution. Although most of the other works encode the evaluation $x = 0$ as the nonexistence of X in the solution, we will use the negation strand NX in order to implement NOT gate simply.

In order to implement DNA logic circuits, we will use the following three basic *abstract-level* chemical reactions:

1. AND reaction: $A \wedge B \rightarrow C$ — the strand C is produced if and only if both of the strands A and B exist in the solution.
2. OR reaction — $A \vee B \rightarrow C$ — the strand C is produced if and only if either of the strands A or B exists in the solution.
3. PROPAGATE (PROP) reaction : $A \rightarrow B$ — the strand B is produced if and only if the strand A exists in the solution.

When we want to construct a chemical reaction system which utilizes any given boolean function, it suffices to devise chemical implementation of logic gates, AND, OR, and NOT. Then, it is easy to construct AND, OR, and NOT gates using the three basic reactions above.

1. AND gate construction — Consider an AND gate with input variables a and b and with an output variable c . Then, we will prepare the strands A , NA for the variable a , B , NB for the variable b , C , NC for the variable c . It is easy to see that the AND and OR reactions, $A \wedge B \rightarrow C$ and $NA \vee NB \rightarrow NC$, implement the AND gate.
2. OR gate construction — Consider an OR gate with input variables a and b and with an output variable c . Then, we will prepare the strands A , NA for the variable a , B , NB for the variable b , C , NC for the variable c . It is easy to see that the AND and OR reactions, $A \vee B \rightarrow C$ and $NA \wedge NB \rightarrow NC$, implement the OR gate.
3. NOT gate construction — Consider a NOT gate with an input variable a and with an output variable b . Then, we will prepare the strands A , NA for the variable a ,

B , NB for the variable b . It is easy to see that the PROP reactions, $A \rightarrow NB$ and $NA \rightarrow B$, implement the NOT gate.

In the rest of this section, we will propose a method to implement AND, OR and PROP reactions with the use of DNA polymerase and nicking enzyme. We assume that we will use a nicking enzyme which recognizes a double stranded DNA sequences $5' - R - 3'$ and $3' - R^* - 5'$, and cleave the 3' end of the lower strand R^* only.

4.1 AND reaction: $A \wedge B \rightarrow C$

We will explain the construction of AND reaction which, with the existence of sufficient amounts of input DNA single strands A and B in a solution, outputs a single stranded DNA sequence C . Principal molecule of this re-

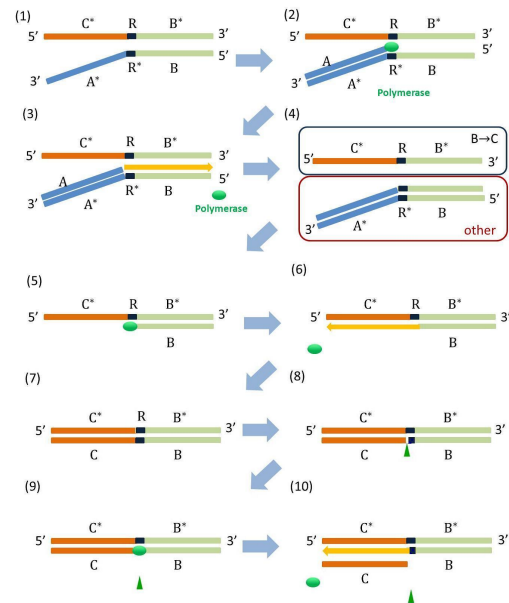


Fig. 2: AND Reaction

action, called *AND complex*, consists of two DNA strands $5' - C^* - R - B^* - 3'$ and $5' - B - R^* - A^* - 3'$ with B and B^* hybridized at its initial state (Fig.2(1)). Let us consider the situation where sufficient amounts of DNA strands A and B exist in a solution. At first, A hybridizes to A^* of the AND complex (Fig.2(2)). Then, DNA polymerase binds to this double stranded part (consisting of A and A^*) and elongates the sequence A until it reaches to the 5'-end of the sequence $5' - B - R^* - A^* - 3'$ (Fig.2(3)). This makes the double stranded part (consisting of B and B^*) detached from the AND complex and we will have DNA strand $5' - C^* - R - B^* - 3'$ in the solution (Fig.2(4)). Then, the strand B hybridizes to B^* of $5' - C^* - R - B^* - 3'$ (Fig.2(5)), and DNA polymerase binds to this double stranded part (consisting of B and B^*) and elongates the

sequence B until it reaches to the 5'-end of the sequence $5' - C^* - R - B^* - 3'$ (Fig.2(6)). This complex is recognized by a nicking enzyme at the site R and R^* , and it cleaves the strand R^* (Fig.2(8)). DNA polymerase, while pushing the strand C away, elongates the strand $5' - B - R^* - 3'$ again until it reaches to the 5'-end of the sequence $5' - C^* - R - B^* - 3'$ and the output strand C is released. Repeating the process (8), (9) and (10), the output strand C is amplified and released.

4.2 OR reaction — $A \mid B \rightarrow C$

We will explain the construction of OR reaction which, with the existence of sufficient amounts of input DNA single strands A or B in a solution, outputs a single stranded DNA sequence C. Principal molecule of this re-

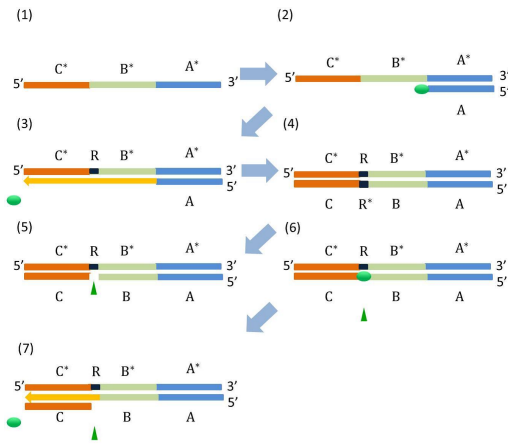


Fig. 3: OR Reaction

action, called *OR complex*, consists of single DNA strand $5' - C^* - R - B^* - A^* - 3'$ at its initial state (Fig.3(1)). Let us consider the situation where sufficient amounts of DNA strands A or B exist in a solution. At first, A or B hybridizes to A^* or B^* of the OR complex (Fig.3(2)). Fig.3 describes the case of A hybridizing to A^* . Then, DNA polymerase binds to this double stranded part and elongates the sequence A until it reaches to the 5'-end of the sequence $5' - C^* - R - B^* - A^* - 3'$ (Fig.3(3),(4)). Next, a nicking enzyme recognizes R and R^* and cleaves the strand R^* (Fig.3(5)). DNA polymerase, while pushing the strand C away, elongates the strand $5' A - B - R^* - 3'$ again until it reaches to the 5'-end of the sequence $5' - C^* - R - B^* - A^* - 3'$ and the output strand C is released. Repeating the process (5), (6) and (7), the output strand C is amplified and released.

4.3 PROP reaction : $A \rightarrow B$

In this subsection, we will explain the construction of PROP reaction which, with the existence of sufficient

amounts of input DNA single strands A in a solution, outputs a single stranded DNA sequence B.

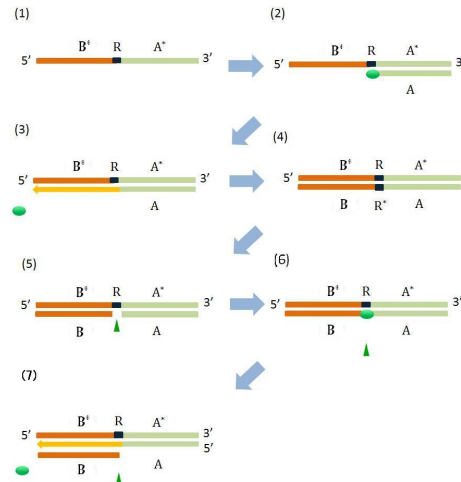


Fig. 4: PROP Reaction

Principal molecule of PROP reaction, called PROP complex, consists of single DNA strand $5' - B^* - R - A^* - 3'$ at its initial state (Fig.4(1)). Let us consider the situation where sufficient amounts of DNA strands A exist in a solution. At first, A hybridizes to A^* of the PROP complex (Fig.4(2)). Then, DNA polymerase binds to this double stranded part and elongates the sequence A until it reaches to the 5'-end of the sequence $5' - B^* - R - A^* - 3'$ (Fig.4(3),(4)). Next, a nicking enzyme recognizes R and R^* and cleaves the strand R^* (Fig.4(5)). DNA polymerase, while pushing the strand B away, elongates the strand $5' - A - R^* - 3'$ again until it reaches to the 5'-end of the sequence $5' - B^* - R - A^* - 3'$ and the output strand B is released. Repeating the process (5), (6) and (7), the output strand B is amplified and released.

5. Mathematical Model of Chemical Reaction Networks

In this section, we will explain the mathematical model of chemical reaction networks for the proposed combinatorial DNA circuits. We only give the explanation for the case of AND reaction, since the models of other basic operations, OR and PROP, are almost similar to that of AND reaction and can be obtained easily.

5.1 Chemical Reaction Network of AND Reaction

For any rate constant k , by k' we denote the rate constant of the reverse reaction of the reaction corresponding to k . Parameters k_{poly}^A , k_{poly}^B and k_{poly}^C are rate constants of polymerase reaction elongating the strands, A, B, and

C, respectively. The parameter k_{nick} is the rate constant for nicking enzyme to cleave strands.

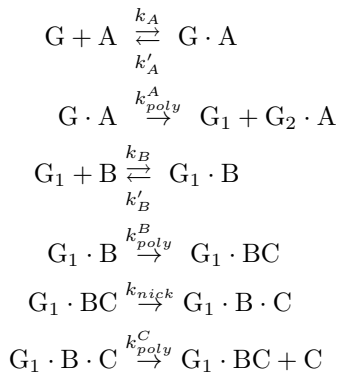


Fig. 5: Full Reaction Network of AND Gate

Chemical reaction network of an *abstract-level* AND reaction is described in Figure 5, where AND complex, its upper strand, and its lower strand are represented as G , G_1 , and G_2 , respectively.

5.2 Differential Equations of AND Reaction

Differential equation system to mathematically model and simulate the AND reaction is given in Figure 6.

6. Simulation Results

We used the following kinetic parameters for the simulation, which are determined by referring to the data reported in [7].

$$\begin{aligned}
 k_A &= k_B = 2.4 \times 10^7 && (\text{M}^{-1}\text{min}^{-1}) \\
 k'_A &= k'_B = 1.2 \times 10^{-3} && (\text{M}^{-1}\text{min}^{-1}) \\
 k_{poly}^B &= 2.4 \times 10^1 && (\text{min}^{-1}) \\
 k_{poly}^A &= k_{poly}^C = 2.4 \times 10^1 \times 0.8 && (\text{min}^{-1}) \\
 k_{nick} &= 6.0 \times 10^0 && (\text{min}^{-1})
 \end{aligned}$$

6.1 Simple Amplification Network

We first constructed a simple amplification network in Figure 7, whose experimental data were reported in [7], and simulated its behavior. The result is given in Figure 8, where we plot the concentration of the hybridized compound of the input strand A and the PROP complex molecule. The obtained simulation curve approximately coincides with the experimentally obtained fluorescence curve reported in [7], and the saturation time is approximately 4 - 6 minutes in both of the simulation and the experimental data. So, we can conclude that the mathematical model and its parameters given above work well corresponding to the experimental data.

$$\begin{aligned}
 \frac{d[A]}{dt} &= -k_A[G][A] + k'_A[G \cdot A] \\
 \frac{d[B]}{dt} &= -k_B[G_1][B] + k'_B[G_1 \cdot B] \\
 \frac{d[C]}{dt} &= k_{poly}^C[G_1 \cdot B \cdot C] \\
 \frac{d[G]}{dt} &= -k_A[G][A] + k'_A[G \cdot A] \\
 \frac{d[G \cdot A]}{dt} &= k_A[G][A] - k'_A[G \cdot A] - k_{poly}^A[G \cdot A] \\
 \frac{d[G_1]}{dt} &= k_{poly}^A[G \cdot A] - k_B[G_1][B] + k'_B[G_1 \cdot B] \\
 \frac{d[G_2 \cdot A]}{dt} &= k_{poly}^A[G \cdot A] \\
 \frac{d[G_1 \cdot B]}{dt} &= k_B[G_1][B] - k'_B[G_1 \cdot B] - k_{poly}^B[G_1 \cdot B] \\
 \frac{d[G_1 \cdot BC]}{dt} &= k_{poly}^B[G_1 \cdot B] - k_{nick}[G_1 \cdot BC] + \\
 &\quad + k_{poly}^C[G_1 \cdot B \cdot C] - k'_{poly}[G_1 \cdot BC][C] \\
 \frac{d[G_1 \cdot B \cdot C]}{dt} &= k_{nick}[G_1 \cdot BC] - k_{poly}^C[G_1 \cdot B \cdot C]
 \end{aligned}$$

Fig. 6: Full system of differential equations for AND reaction

6.2 Majority Vote Circuit

We applied the proposed method to the construction of a majority vote circuit. The circuit outputs 1 if the number of input variables assigned to 1 is greater than that of input variables assigned to 0. We applied the method to 3-variable case. The circuit is given in Figure 9. For each variable, x , y , z , a , b , c , d , e , we will prepare the DNA strands X , Y , Z , A , B , C , D , E , for representing the evaluation of each variable to 1. Furthermore, the DNA strands NX , NY , NZ , NA , NB , NC , ND , NE are used for representing the evaluation of each variable to 0.

We set the concentration of each input strand X , Y , Z , NX , NY , NZ , to 1.0×10^{-10} (M) if it should exist in the solution, to 0 (M) otherwise. Furthermore, the concentration of the AND-complex and OR-complex at the 1st, 2nd, and 3rd layers are set to 1.0×10^{-9} (M), 1.0×10^{-7} (M), and 5.0×10^{-8} (M), respectively.

Figure 10 gives the simulation result for the case of inputs $x = y = z = 1$. That is, we put 1.0×10^{-10} (M) of X , Y , Z , and 0 (M) of NX , NY , NZ , in the solution. As expected, the concentration of the output strand E grows rapidly, and NE stays at 0 for all the time of the simulation, which correctly simulates the behavior of the majority vote circuit.

For the case of inputs $x = y = z = 0$. That is, we put 1.0×10^{-10} (M) of NX , NY , NZ , and 0 (M) of X , Y , Z , in the solution. As expected, the concentration of the output strand NE grows rapidly, and E stays at 0 for all the time

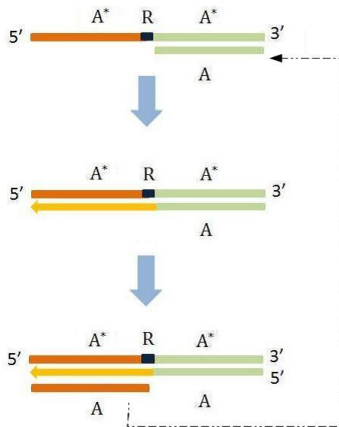


Fig. 7: Simple Amplification Network

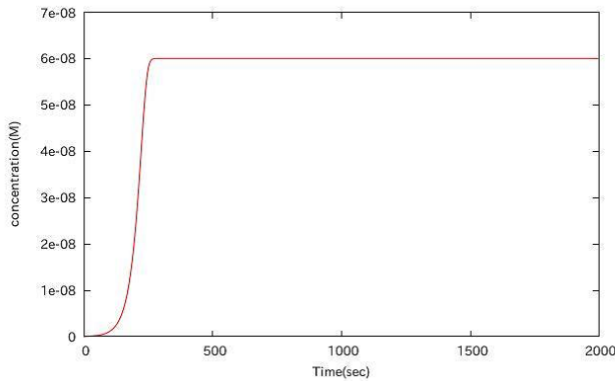


Fig. 8: Simulation of Simple Amplification Network

of the simulation, which correctly simulates the behavior of the majority vote circuit (Fig.11). Note that in Fig. 11, the concentrations of NA and NC are very small, so they almost stay at 0 in the graph.

Figure 12 gives the global view of the simulation result for the case of inputs $x = y = z = 1$. We can verify that the growth of the 3rd layer output is very rapid (at most 10 minutes are enough for 3 steps of logic operation).

On the other hand, the output strands behavior largely depends on the initial concentrations of principal complex of each gate and each input strand. For instance, if we put 1.0×10^{-10} (M) of the input strands, 1.0×10^{-11} (M) of AND-complex and OR-complex in the solution, then, for the inputs $x = y = z = 1$, the output strand E grows very slowly (See Figure 13). So, design of these concentration parameters are important problem in the future.

7. Conclusions

We proposed a new bio-lab method for constructing logic circuits with DNA molecules, where we use the amplification

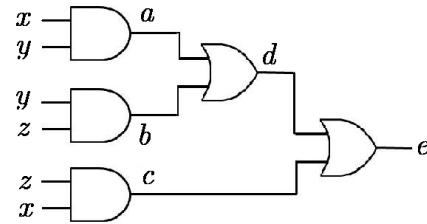


Fig. 9: Majority Vote Circuit

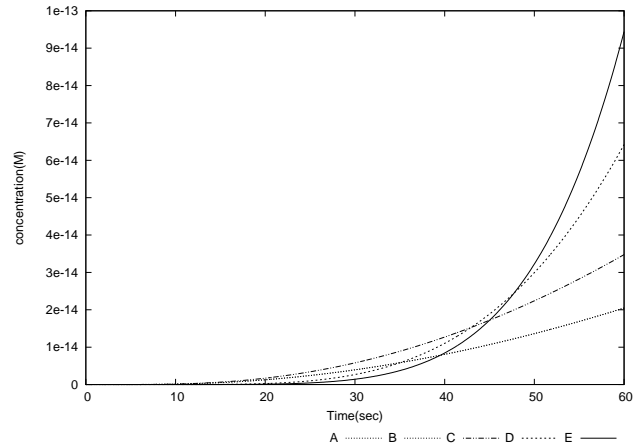


Fig. 10: Simulation of Majority Vote Circuit (1)

system based on a nicking enzyme and a DNA polymerase similar to [13], [6], and [7]. The proposed method has some good properties as computational devices that it is more time-efficient than the strand displacement approaches ([17][15][18], etc.) and that the output of each DNA logic gate can be amplified, and so, the scalability and the feasibility of the proposed system is better than the previous methods.

However, in order to increase the amount of the output molecule, the concentration of each gate complex should be carefully designed as is discussed in section 6.2. Furthermore, the proposed method has a problem that the amplification of the output strands continues until DNA polymerase uses up substrates. So, we need a bio-lab method for stopping or inhibiting the amplification process in this framework. An idea is to use inhibitor strands and exonuclease as in [7]. All of these issues are our future research topics.

Acknowledgement

The second author is supported in part by Grant-in-Aid for Scientific Research (C) No.22500010, Japan Society for the Promotion of Science.

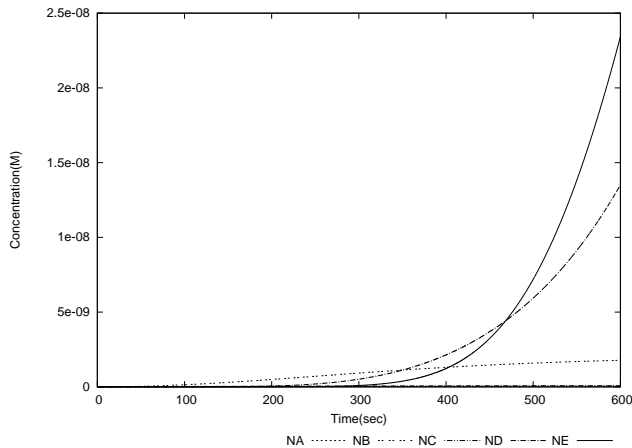


Fig. 11: Simulation of Majority Vote Circuit (2)

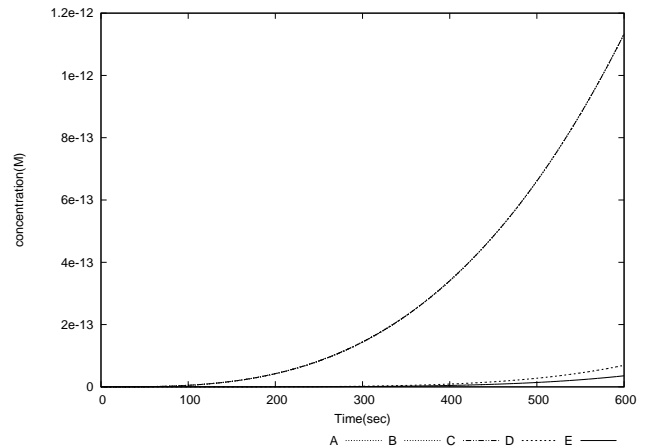


Fig. 13: Simulation of Majority Vote Circuit (4)

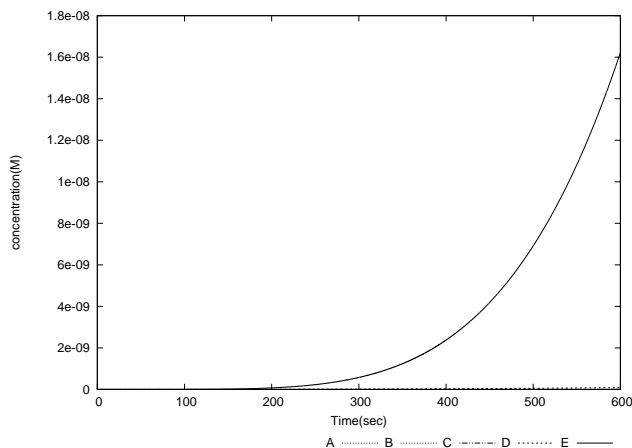


Fig. 12: Simulation of Majority Vote Circuit (3)

References

- [1] L. Adleman, Molecular Computation of Solutions to Combinatorial Problems, *Science* **266**, pp.1021-1024, 1994.
- [2] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, E. Shapiro, Programmable and autonomous computing machine made of biomolecules, *Nature* **414**, pp.430-434, 2001.
- [3] M. Hagiya, M. Arita, D. Kiga, K. Sakamoto, S. Yokoyama, Towards parallel evaluation of Boolean μ -formulas with molecules, In *DNA Based Computers III* (American Mathematical Society), pp.57-72, 1999.
- [4] R. Lipton, DNA solution of hard computational problem, *Science* **268**, pp.542-545, 1995.
- [5] D. Zhang, E. Winfree, K. Lund, A. Manzo, N. Dabby, N. Michelotti, A. Johnson-Buck, J. Nangreave, S. Taylor, R. Pei, M. Stojanovic, N. Walter, E. Winfree, H. Yan, Molecular robots guided by prescriptive landscapes, *Nature*, **465**, pp.206-210, 2010.
- [6] D. Matsuda, M. Yamamura, Cascading Whiplash PCR with a Nicking Enzyme, DNA Computing — Proc. of International Workshop on DNA-Based Computers, DNA8 (2002), Lecture Notes in Computer Science, pp.38-46, 2003.
- [7] K. Montagne, R. Plasson, Y. Sakai, T. Fujii, Y. Rondelez, Programming an *in vitro* RNA oscillator using a molecular networking strategy, *Molecular Systems Biology*, **7**, Article Number 466, 2011.
- [8] T. Omabegho, R. Sha, N. Seeman, A Bipedal DNA Brownian Motor with Coordinated Legs, *Science* **324**, pp.67-71, 2009.
- [9] L. Qian, D. Soloveichik, E. Winfree, Efficient Turing-universal computation with DNA polymers, *Lecture Notes in Computer Science* **6518**, pp.123-140, 2011.
- [10] P. Rothemund, Folding DNA to create nanoscale shapes and patterns, *Nature* **440**, pp.297-302, 2006.
- [11] G. Seelig, D. Soloveichik, D. Zhang, E. Winfree, Enzyme-free nucleic acid logic circuits, *Science*, **314**, pp.1585-1588, 2006.
- [12] D. Soloveichik, G. Seelig, E. Winfree, DNA as a Universal Substrate for Chemical Kinetics, *Proc. Natl. Acad. Sci. USA*, **107**, pp.5393-5398, 2010.
- [13] G. Walker, M. Little, J. Nadeau, D. Shank, Isothermal *in vitro* amplification of DNA by a restriction enzyme/DNA polymerase system, *Proc. Natl. Acad. Sci. USA*, **89**, pp.392-396, 1992.
- [14] E. Winfree, F. Liu, L. Wenzler, N. Seeman, Design and self-assembly of two-dimensional DNA crystals, *Nature* **394**, pp.539-544, 1998.
- [15] P. Yin, H. Choi, C. Calvert, N. Pierce, Programming biomolecular self-assembly pathways, *Nature*, **451**, pp.318-322, 2008.
- [16] B. Yurke, A.J. Turberfield, A.P. Mills, F.C. Simmel, J. Neumann, A DNA-fuelled molecular machine made of DNA. *Nature* **406**, pp.605-608, 2000.
- [17] D. Zhang, A. Turberfield, B. Yurke, E. Winfree, Engineering Entropy-Driven Reactions and Networks Catalyzed by DNA, *Science*, **318**, pp.1121-1125, 2007.
- [18] D. Zhang, E. Winfree, Control of DNA Strand Displacement Kinetics Using Toehold Exchange, *J. of American Chemical Society*, **131**, pp.17303-17314, 2009.

An Improved Shift Strategy for the Modified Discrete Lotka-Volterra with Shift Algorithm

Masami Takata¹, Takumi Yamashita², Akira Ajisaka², Kinji Kimura² and Yoshimasa Nakamura²

¹Graduate School of Humanities and Sciences, Nara Women's University, Nara, Nara, JAPAN

²Graduate School of Informatics, Kyoto University, Kyoto, Kyoto, JAPAN

Abstract—We propose a new mathematical shift strategy for the modified discrete Lotka-Volterra with shift (mdLVs) algorithm. The mdLVs algorithm computes the singular values of bidiagonal matrices. It is known that the convergence of the mdLVs algorithm is accelerated when the shift is close to and less than the square of the smallest singular value of the input matrix. In the original mdLVs algorithm, the Johnson bound is adopted. Our improved mdLVs algorithm combines the Gerschgorin-type bound, the Kato-Temple bound, the Laguerre shift, and the generalized Newton shift. For different combinations, we discuss the computational time and number of iterations.

Keywords: singular value decomposition, Gerschgorin-type bound, Kato-Temple bound, Laguerre shift, Newton shift, mdLVs

1. Introduction

Singular value decomposition (SVD) is one of the most important matrix operations in numerical algebra, and it plays an important role in fields such as data search systems [5] and image processing [13].

Several SVD algorithms are composed by computing singular values and singular vectors. The modified discrete Lotka-Volterra with shift (mdLVs) algorithm [3], [4], [14], [15] computes singular values; its speed and relative accuracy are excellent.

The mdLVs iteration involves the computation of shifts. It is known that the convergence of the mdLVs algorithm is accelerated when the shift is close to and less than the square of the smallest singular value of the input matrix. The Integrable-SVD [3], [14], [15], [16], for which a library has been developed [2], includes the original mdLVs algorithm. It uses the Johnson bound [6] as shift strategy. This bound can compute a sharper bound among various shift strategies. However, since $2M - 1$ square roots must be found, the Johnson bound has a large computational time. Here, M is the dimension size of the input matrix. Therefore, a fast and mathematically rigorous shift strategy is needed.

In this paper, we improve the shift strategy for the mdLVs algorithm. First, we compute a lower bound of the smallest singular value from the Gerschgorin theorem [1]. Let us call this bound the Gerschgorin-type bound. Since this is always weaker than the Johnson bound after enough number of iterations, we then consider the Kato-Temple bound [7].

We compare the two bounds to determine a shift for the mdLVs algorithm. In some cases, the Laguerre shift [11] or the generalized Newton shift [9], [10] instead of the Gerschgorin-type bound is adopted. The improved shift can be computed with M square-root operations.

In Section 2, we explain the mdLVs algorithm. In Section 3, we introduce the Johnson bound. In Section 4, we describe the improved shift strategy for the mdLVs algorithm. In Section 5, we present numerical experiments and confirm that the mdLVs algorithm with the new strategy is faster than the original algorithm.

2. Modified discrete Lotka-Volterra with shift algorithm

In Section 2.1, we give a summary of the singular value computation based on the discrete Lotka-Volterra (dLV) system. In Section 2.2, we outline the mdLVs algorithm. In Section 2.3, we briefly describe the implementation of the mdLVs algorithm.

2.1 Singular value computation based on the discrete Lotka-Volterra system

In mathematical biology, the Lotka-Volterra (LV) system is known as a fundamental prey-predator model. In some cases, the LV system is a completely integrable dynamical system with explicit solutions and sufficiently many conservation laws. A time discretization

$$u_k^{(n+1)} = \frac{1 + \delta^{(n)} u_{k+1}^{(n)}}{1 + \delta^{(n+1)} u_{k-1}^{(n+1)}} u_k^{(n)} \quad (1)$$

of the LV system is known (cf. [3]). This system also has an explicit solution and many conservation laws. Therefore, it is called the integrable dLV system. Here, k ($k = 1, 2, \dots, 2M - 1$) indicates the k th species, the discrete time n ($n = 0, 1, 2, \dots$) corresponds to the iteration number of the algorithm, $u_k^{(n)}$ is the value of u_k at n , and the arbitrary nonzero number $\delta^{(n)}$ is a discrete step-size. Let the initial value $u_k^{(0)}$ be positive. In the case where $\delta^{(n)} > 0$, any subtraction and division by zero do not occur in Eq.(1) and $u_k^{(n)}$ is always positive. Consequently, cancellation and numerical instability do not occur. Note that we do not need to treat negative numbers in singular value computations.

The boundary condition and the initial condition are given by

$$u_0^{(n)} \equiv 0, \quad u_{2M}^{(n)} \equiv 0, \quad (2)$$

$$u_k^{(0)} = \frac{(b_k)^2}{1 + \delta^{(0)} u_{k-1}^{(0)}}. \quad (3)$$

respectively. Here, $b_{2i-1} (> 0)$ and $b_{2i} (> 0)$ ($i: 1 \leq i \leq M$) are the diagonal and upper-subdiagonal elements, respectively, of the $M \times M$ bidiagonal matrix B .

$$B = \begin{pmatrix} b_1 & b_2 & & & & \\ & b_3 & b_4 & & & \\ & & \ddots & \ddots & & \\ & & & b_{2(M-1)-1} & b_{2(M-1)} & \\ & & & & b_{2M-1} & \end{pmatrix}. \quad (4)$$

When $n \rightarrow \infty$, $u_{2i-1}^{(n)}$ and $u_{2i}^{(n)}$ converge to the square of the i th singular value σ_i and 0, respectively. Thus the dLV system gives rise to a stable scheme for computing the singular values [3].

2.2 Improved speed via a shifted discrete Lotka-Volterra scheme

The mdLVs algorithm, the integrable dLV system with a shift, can compute the singular values more quickly. The mdLVs algorithm is as follows [4].

Let us introduce new elements $w_k^{(n)}$ and $v_k^{(n)}$ by

$$w_k^{(n)} = u_k^{(n)} (1 + \delta^{(n)} u_{k-1}^{(n)}), \quad (5)$$

$$v_k^{(n)} = u_k^{(n)} (1 + \delta^{(n)} u_{k+1}^{(n)}). \quad (6)$$

By Eq.(3), the initial $w_k^{(0)}$ is just b_k^2 . The shifted integrable dLV system is defined by adding to Eq.(1) a shift Θ at the n th iteration defined as $0 \leq \Theta < \sigma_{min}^2$ where σ_{min} is the smallest singular value of B . This gives

$$\begin{aligned} w_{2i-1}^{(n+1)} &= v_{2i-1}^{(n)} + v_{2i-2}^{(n)} - w_{2i-2}^{(n+1)} - \Theta, \\ w_{2i}^{(n+1)} &= v_{2i-1}^{(n)} v_{2i}^{(n)} / w_{2i-1}^{(n+1)}. \end{aligned} \quad (7)$$

In general, the convergence is accelerated by increasing Θ . However, since the positivity of $u_k^{(n)}$ may be destroyed by a larger Θ at the n th iteration, this causes a numerical instability. It is proved in [4] that $u_k^{(n)} > 0$ if and only if $0 \leq \Theta < \sigma_{min}^2$. Hence, we can determine the shift Θ for estimating σ_{min} .

2.3 Algorithm for singular value computation based on the Lotka-Volterra system

Each iteration in the mdLVs algorithm is as follows.

- 1) Calculate $u_k^{(n)}$ from $w_k^{(n)}$ via Eq.(5).
- 2) Calculate $v_k^{(n)}$ from $u_k^{(n)}$ via Eq.(6).
- 3) Calculate the shift Θ at the n th iteration.
- 4) Check Θ and calculate $w_k^{(n+1)}$ accordingly.

- If Θ is valid, calculate $w_k^{(n+1)}$ from $v_k^{(n)}$ via Eq.(7).
- Otherwise, $w_k^{(n+1)} = v_k^{(n)}$.

- 5) If $w_{2i}^{(n+1)}$ is much smaller than $w_{2i-1}^{(n+1)}$, perform SPLIT or a deflation of the dimension as described in [12].

SPLIT, which divides the matrix into two parts, and the deflation are defined.

The arrays of the algorithm are calculated as follows. In Step 1), the array $U = (u_1^{(n)}, u_2^{(n)}, \dots, u_{2M-1}^{(n)})$ is calculated from the array $W = (w_1^{(n)}, w_2^{(n)}, \dots, w_{2M-1}^{(n)})$. Since we do not keep the data for each n , each array is a one-dimensional array corresponding to the subscript. In Step 2), the array $V = (v_1^{(n)}, v_2^{(n)}, \dots, v_{2M-1}^{(n)})$ is calculated from U . In Step 3), the shift Θ at the n th iteration is calculated from V . Using the valid Θ , we overwrite W with V in Step 4).

In the loops of Steps 1) and 2), U and V are updated in ascending order of k . To update $u_k^{(n)}$, we use $w_k^{(n)}$ and $u_{k-1}^{(n)}$ in Step 1). To update $v_k^{(n)}$, we need $u_k^{(n)}$ and $u_{k+1}^{(n)}$ in Step 2). To calculate the shift, we use V instead of B in Step 3), since the upper bidiagonal matrix at the n th iteration is expressed using V . For the update $w_k^{(n+1)}$ from $v_k^{(n)}$ in Step 4), W is updated in ascending order of k .

3. Johnson bound

The theorem for the Johnson bound is a corollary of the Gerschgorin circle theorem for $\frac{(B^\top + B)}{2}$.

$$\frac{B^\top + B}{2} = \begin{pmatrix} b_1 & \frac{b_2}{2} & & & & \\ \frac{b_2}{2} & b_3 & \frac{b_4}{2} & & & \\ & \ddots & \ddots & \ddots & & \\ & & & b_{2(M-1)-1} & \frac{b_{2(M-1)}}{2} & \\ & & & \frac{b_{2(M-1)}}{2} & b_{2M-1} & \end{pmatrix}. \quad (8)$$

Since the singular values in B are equal to those in B^\top , the Johnson bound for the smallest singular value of an upper bidiagonal matrix B is given as the following inequality:

$$\sigma_{min} \geq \min_{1 \leq i \leq M} \left[b_{2i-1} - \frac{1}{2} (b_{2i} + b_{2i-2}) \right], \quad (9)$$

where $b_0 = b_{2M} = 0$.

In the mdLVs algorithm, b_k becomes $\sqrt{v_k^{(n)}}$ at the n th iteration, and the shift Θ is defined as $0 \leq \Theta < \sigma_{min}^2$. Therefore, the Johnson shift Θ_J is computed using the

4.2 Kato-Temple bound

Let A be a real symmetric matrix and x be a real vector. Let $\rho = x^T A x$ be its Rayleigh quotient with $x^T x = 1$. For a given eigenvalue λ of A , we introduce the Kato-Temple inequality.

Let us assume that the open interval $(\underline{\lambda}, \bar{\lambda})$ includes an eigenvalue λ of A as well as the Rayleigh quotient ρ and that it does not include any other eigenvalues. Then, we have an inequality given by the following theorem:

$$\rho - \frac{\varepsilon^2}{\bar{\lambda} - \rho} \leq \lambda \leq \rho + \frac{\varepsilon^2}{\rho - \underline{\lambda}}, \tag{16}$$

where $\varepsilon^2 = \|Ax - \rho x\|_2^2$.

In the following discussion, we consider an application of the Kato-Temple inequality for the smallest eigenvalue λ_{\min} of the symmetric positive definite tridiagonal matrix of the form $A = BB^T$, except $M \leq 2$. The eigenvalues λ_i of A satisfy $0 < \lambda_M < \lambda_{M-1} < \dots < \lambda_1$. Let $A^{(i)}$ be a $i \times i$ submatrix of A such that $|A^{(i)}|$ is the i th principal minor determinant of A . Let $\{\lambda_j^{(i)}\}_{j=1, \dots, i}$ be a set of eigenvalues of $A^{(i)}$. Note that $A = A^{(M)}$ and $\lambda_j = \lambda_j^{(M)}$. The separation theorem (interlacing property) [8] for eigenvalues of symmetric tridiagonal matrices is $\lambda_i^{(i)} < \lambda_{i-1}^{(i-1)} < \lambda_{i-1}^{(i)} < \lambda_{i-2}^{(i-1)} < \dots < \lambda_1^{(i)}$ for $i = 1, \dots, M$.

Define a sequence $\{t_i\}$ by $t_1 = b_1^2 + b_2^2$, $t_{i+1} = b_{2(i+1)-1}^2 + b_{2(i+1)}^2 - b_{2i-1}b_{2(i-1)}/t_i$ for $i = 1, 2, \dots, M - 2$, $t_M = b_{2M-1}^2 - b_{2M-1}b_{2(M-1)}/t_{M-1}$. Since $|A^{(i)}| = t_1 t_2 \dots t_i > 0$, we see that $t_i > 0$. By definition, we have

$$\begin{aligned} b_{2M-1}^2 > t_M &= \frac{|A^{(M)}|}{|A^{(M-1)}|} \\ &= \frac{\lambda_1^{(M)} \dots \lambda_M^{(M)}}{\lambda_1^{(M-1)} \dots \lambda_{M-1}^{(M-1)}} \\ &= \frac{\lambda_1^{(M)}}{\lambda_1^{(M-1)}} \dots \frac{\lambda_{M-1}^{(M)}}{\lambda_{M-1}^{(M-1)}} \lambda_M^{(M)} \\ &> \lambda_M^{(M)} = \lambda_M. \end{aligned} \tag{17}$$

Now we have a candidate for the Rayleigh quotient ρ such that $\lambda_M < \rho$. Let us choose the unit vector x for

$$x = (0, \dots, 0, 1)^T. \tag{18}$$

The Rayleigh quotient is then given by

$$\rho = x^T A x = b_{2M-1}^2 (> \lambda_M). \tag{19}$$

Finally, we consider how to choose the right endpoint $\bar{\lambda}$ of the open interval $(\underline{\lambda}, \bar{\lambda})$ including λ_m , and not including any other eigenvalues. The separation theorem says that a good

bound of the smallest eigenvalue $\lambda_{m-1}^{(m-1)}$ of the submatrix

$$A^{(M-1)} = \begin{pmatrix} b_1^2 + b_2^2 & & & & \\ b_3 b_2 & b_3^2 + b_4^2 & & & \\ & \ddots & \ddots & \ddots & \\ & & b_{2M-3} b_{2M-4} & b_{2M-3}^2 + b_{2M-2}^2 & \\ & & & & \end{pmatrix}. \tag{20}$$

may give $\bar{\lambda}$ such that the assumption $\lambda_M < \rho < \bar{\lambda}$ is satisfied. In this case, we obtain the Kato-Temple bound Θ_K of the smallest eigenvalue λ_M of A from the Kato-Temple inequality. The Kato-Temple bound Θ_K is given as follows:

$$\begin{aligned} \Theta_K &= \rho - \frac{\varepsilon^2}{\bar{\lambda} - \rho} \\ &= b_{2M-1}^2 - \frac{\|Ax - \rho x\|_2^2}{\bar{\lambda} - b_{2M-1}^2} \\ &= b_{2M-1}^2 - \frac{b_{2M-1}^2 b_{2(M-1)}^2}{\bar{\lambda} - b_{2M-1}^2} \\ &\leq \lambda_M. \end{aligned} \tag{21}$$

The bound $\Theta^{(M-1)}$ of $\lambda_{M-1}^{(M-1)}$ should be computed using the original bound, for example, the Gerschgorin-type bound and the generalized Newton bound. We call such a bound an auxiliary bound. If the assumption $\Theta^{(M-1)} > b_{2M-1}^2 (= \rho)$ is satisfied, we obtain the Kato-Temple bound by Eq.(21) where $\bar{\lambda} = \Theta^{(M-1)}$.

4.3 Laguerre shift

Let us set $J_1^{(-)} = \text{trace}(BB^T)^{-1}$ and $J_2^{(-)} = \text{trace}((BB^T)^2)^{-1}$. The Laguerre shift Θ_L is defined as follows [11]:

$$\Theta_L = \frac{1}{J_1^{(-)}} \cdot \frac{M}{1 + \sqrt{(M-1) \left(M \frac{J_2^{(-)}}{(J_1^{(-)})^2} - 1 \right)}} > 0. \tag{22}$$

Theoretically, $\left(M \frac{J_2^{(-)}}{(J_1^{(-)})^2} - 1 \right)$ is positive, however computationally, the value is occasionally negative.

When the iteration number n is small, the Gerschgorin-type bound may be non-positive. On the other hand, in almost cases, the Laguerre shift Θ_L becomes positive, since $\left(M \frac{J_2^{(-)}}{(J_1^{(-)})^2} - 1 \right)$ is non-negative. However, computationally, if $a_{i,i} - (a_{i,i-1} + a_{i,i+1}) \leq 0$ for some i ($(1-\kappa)M \leq i \leq M$), then the Laguerre shift is not so close to the smallest singular value when the iteration number is small. Here $\kappa \in (0, 1)$ is a constant. Therefore, if all the expressions $a_{i,i} - (a_{i,i-1} + a_{i,i+1})$ are positive for $(1-\kappa)M \leq i \leq M$, we calculate the Laguerre shift Θ_L instead of returning zero derived from the Gerschgorin theorem. Experimentally, $\kappa = 0.02$ is the best choice.

4.4 Generalized Newton shift

The generalized Newton bound of the smallest singular value σ_{min} of B is given as follows [8]:

$$\begin{aligned}\Theta_p^{(M)} &= (\text{trace}(B^T B)^{-p})^{-\frac{1}{2p}} \\ &= \frac{1}{\left(\frac{1}{\sigma_1^{2p}} + \dots + \frac{1}{\sigma_M^{2p}}\right)^{\frac{1}{2p}}} > 0,\end{aligned}\quad (23)$$

where p is an arbitrary positive integer. These bounds have the properties listed in [9] [10].

$$\Theta_1^{(M)} < \Theta_2^{(M)} < \dots < \sigma_M, \quad (24)$$

$$\lim_{p \rightarrow \infty} \Theta_p^{(M)} = \sigma_M. \quad (25)$$

Then, $(\Theta_p^{(M)})^2$ ($p = 1, 2, \dots$) can be used. Let us call $(\Theta_p^{(M)})^2$ the generalized Newton shift of order p .

The generalized Newton shift $(\Theta_p^{(M)})^2$ can be computed within $O(Mp^2)$ flops using a recurrence-relation formula. This proof for the computational cost should be discussed in another paper, on which T. Yamashita, K. Kimura, and Y. Nakamura are working.

4.5 New shift strategy and its implementation

In this section, we improve the shift strategy from that using the Johnson bound.

In most microprocessors, a square-root operation takes longer time than addition and multiplication operations. Therefore, the number of square-root operations should be reduced.

The Johnson bound requires $2M - 1$ times of square-root operations. On the other hand, the Gerschgorin-type bound needs just $M - 1$ times of square-root operations. Consequently, the Gerschgorin-type bound is expected as a measure to improve the shift strategy with the Johnson bound. However, we have to consider the following possibilities. The Gerschgorin-type bound Θ_G may be smaller than the Johnson bound Θ_J . Especially, after enough number of iterations, since it holds

$$\Theta_J = \left(\sqrt{v_{2M-1}^{(n)}} - \frac{1}{2} \sqrt{v_{2(M-1)}^{(n)}} \right)^2, \quad (26)$$

$$\Theta_G = v_{2M-1}^{(n)} - \sqrt{v_{2M-1}^{(n)} v_{2(M-1)}^{(n)}}, \quad (27)$$

in the mdLVs algorithm, we have $\Theta_J > \Theta_G$.

Furthermore, the Gerschgorin-type bound may give a non-positive value. Then, we devise computation of shift as follows. If the Gerschgorin-type bound gives a positive value, we compute the Kato-Temple bound and adopt the square of larger bound between these two bounds as the shift. If the Gerschgorin-type bound gives a non-positive value, we compute the Laguerre shift or take shift zero according to the condition described in the Section 4.3. If the Laguerre shift

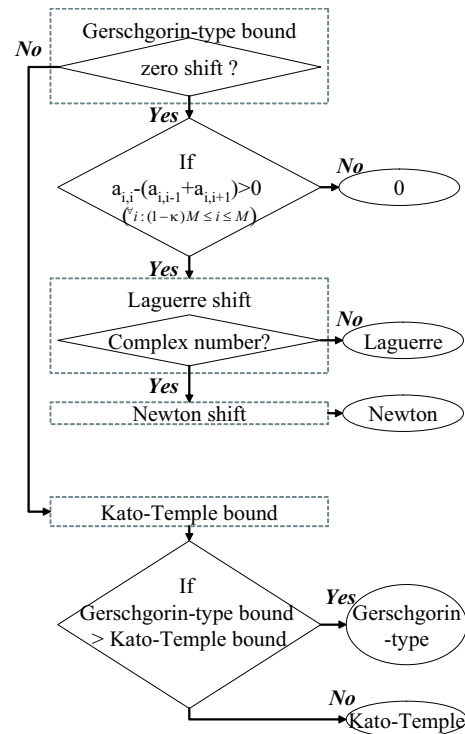


Fig. 1: A flowchart of an improvement of shift strategy.

numerically gives a complex number since $\left(M \frac{J_2^{(-)}}{(J_1^{(-)})^2} - 1\right)$ is occasionally negative, then we compute the generalized Newton shift.

Figure 1 shows a flowchart of the improved shift strategy. An implementation of the strategy is given in Algorithms 2 and 3.

5. Numerical experiments

To confirm the performance of the improved shift strategy, the computational time and number of iterations for the mdLVs algorithm in [2] with four shift strategies. The shifts are as follows:

- SHIFT(J): Johnson bound
- SHIFT(G): Gerschgorin-type bound
- SHIFT(GK): SHIFT(G) and Kato-Temple bound
- SHIFT(GKL): SHIFT(GK), Laguerre and generalized Newton shifts.

We use a computer with an Intel(R) Xeon(R) X5570@2.93GHz CPU and 32GB of memory. Fedora 13 is installed on this computer. The input matrices B are bidiagonal and the diagonal and subdiagonal elements of B are set randomly in an interval $[0, 1]$. The dimension is 30000, We set $\kappa = 0.02$ and $\delta^{(n)} = 1$, respectively.

Table 1 gives the average computational time and number of iterations for 100 matrices.

Algorithm 2 Implementation of the recommended shift strategy (first half)

```

1:  $T_1 \leftarrow \sqrt{v_{2M-1}^{(n)} * v_{2(M-1)}^{(n)}}$ 
2:  $S_0 \leftarrow v_{2M-1}^{(n)} - T_1$ 
3: if  $S_0 \leq 0$  then
4:   return 0
5: end if
6:  $T_2 \leftarrow v_{2(M-1)-1}^{(n)} + v_{2(M-2)}^{(n)}$ 
7:  $T_3 \leftarrow \sqrt{v_{2(M-1)-1}^{(n)} * v_{2(M-2)}^{(n)}}$ 
8:  $S \leftarrow T_2 - T_3$ 
9:  $S_1 \leftarrow S - T_1$ 
10: if  $S_1 \leq 0$  then
11:   if  $M - 1 \geq (1 - \kappa)M$  then
12:     return 0
13:   else
14:     goto Algo.3
15:   end if
16: end if
17: for  $i = M - 2$  to 2 by  $-1$  do
18:    $T_2 \leftarrow v_{2i-1}^{(n)} + v_{2i}^{(n)} - T_3$ 
19:    $T_3 \leftarrow \sqrt{v_{2i-1}^{(n)} * v_{2i}^{(n)}}$ 
20:    $T_2 \leftarrow T_2 - T_3$ 
21:   if  $T_2 \leq 0$  then
22:     if  $i \geq (1 - \kappa)M$  then
23:       return 0
24:     else
25:       goto Algo.3
26:     end if
27:   end if
28:    $S \leftarrow \min(S, T_2)$ 
29: end for
30:  $T_2 \leftarrow v_1^{(n)} + v_2^{(n)} - T_3$ 
31: if  $T_2 \leq 0$  then
32:   goto Algo.3
33: end if
34:  $S \leftarrow \min(S, T_2)$ 
35:  $T_3 \leftarrow S - v_{2M-1}^{(n)}$ 
36:  $S \leftarrow \min(S, S_0, S_1)$ 
37: if  $T_3 > 0$  then
38:    $S \leftarrow \max(S, v_{2M-1}^{(n)} * (1.0 - v_{2(M-1)}^{(n)}/T_3))$ 
39: end if
40: return  $S$ 

```

Algorithm 3 Implementation of the recommended shift strategy (latter half)

```

1:  $J[M] \leftarrow 1.0/v_{2M-1}^{(n)}$ 
2:  $D[M] \leftarrow J[M]$ 
3: for  $i = M - 1$  to 1 by  $-1$  do
4:    $J[i] \leftarrow 1.0/v_{2i-1}^{(n)}$ 
5:    $D[i] \leftarrow J[i] * (v_{2i}^{(n)} * D[i + 1] + 1.0)$ 
6: end for
7:  $R \leftarrow J[1]$ 
8:  $W \leftarrow R * D[1]$ 
9:  $S \leftarrow R$ 
10:  $T \leftarrow W$ 
11: for  $i = 2$  to  $M$  do
12:    $E \leftarrow J[i] * v_{2(i-1)}^{(n)}$ 
13:    $H \leftarrow E * R$ 
14:    $R \leftarrow H + J[i]$ 
15:    $S \leftarrow S + R$ 
16:    $W \leftarrow E * W + (R + H) * D[i]$ 
17:    $T \leftarrow T + W$ 
18: end for
19:  $W \leftarrow M$  :  $M$  is cast into double precision number
20:  $R \leftarrow W * T / (S * S) - 1.0$ 
21: if  $R \geq 0$  then
22:    $W \leftarrow W / (S * (1.0 + \sqrt{(W - 1.0) * R}))$  : compute
   the Laguerre shift  $\Theta_L$ 
23: else
24:    $W \leftarrow \sqrt{1.0/T}$  : compute the generalized Newton
   shift  $(\Theta_2^{(M)})^2$ 
25: end if
26: return  $W$ 

```

Table 1: Computation time and iteration number in each shift

	computation time[sec.]	iteration number
SHIFT(J)	27.61	315021
SHIFT(G)	23.06	368773
SHIFT(GK)	23.09	362114
SHIFT(GKL)	20.78	206941

SHIFT(G) and SHIFT(GK) require more iterations than SHIFT(J) does. This implies that Θ_J tends to be stronger than Θ_G and Θ_K . Thus, the Gerschgorin-type bound itself nor the combination of the Gerschgorin-type bound and the Kato-Temple bound lead to a suitable shift. In spite of much the number of iterations, the computational time of SHIFT(G) and SHIFT(GK) are shorter than that of SHIFT(J). This is because of the numbers of square-root operations in SHIFT(G) and SHIFT(GK) are smaller than that in SHIFT(J). A square-root operation requires relatively large computational time.

On the other hand, SHIFT(GKL) gives better results in both the computational time and the number of iterations than SHIFT(J). In SHIFT(GKL), when the Gerschgorin-type bound returns non-positive value, the Laguerre shift

is computed under the condition shown in Figure 1. We suppose such result attained from utilization of the Laguerre shift. Therefore, we recommend SHIFT(GKL) as a shift strategy for the mdLVs algorithm.

6. Conclusions

In this paper, we have improved the shift strategy for the mdLVs algorithm with the Johnson bound. The improved strategy utilizes the Gerschgorin-type bound, the Kato-Temple bound, the Laguerre shift, and the generalized Newton shift. This improvement takes advantage of less times of square-root operations of the Gerschgorin-type bound than the Johnson bound. There are possibilities that the Gerschgorin-type bound gives a smaller value than the Johnson bound or a non-positive value. Then, we consider the Kato-Temple bound, the Laguerre shift, and the generalized Newton shift. To validate the performance of the improved strategy, we explored the computational time and the number of iterations. The result shows that the improved strategy is efficient.

In future work, we plan to study the relative errors of the computed singular values in the improved shift strategy.

Acknowledgment

This work is partially supported by JSPS Grant-in-Aid for Scientific Research (A) Japan, No.20246027.

References

- [1] S. Gerschgorin, "Über die Abgrenzung der Eigenwerteeiner Matrix," *Izv. Akad.Nauk. USSR Otd. Fiz.-Mat. Nauk*, Vol. 7, pp. 749–754, 1931.
- [2] (2010) I-SVD library. [Online]. Available: <http://www-is.amp.i.kyoto-u.ac.jp/lab/isvd/download/>
- [3] M. Iwasaki and Y. Nakamura, "On the convergence of a solution of the discrete Lotka-Volterra system," *Inverse Problems*, Vol. 18, pp. 1569–1578, 2002.
- [4] M. Iwasaki and Y. Nakamura, "Accurate computation of singular values in terms of shifted integrable schemes," *Japan Journal of Industrial and Applied Mathematics*, Vol. 23, pp. 239–259, 2006.
- [5] F. Jiang, R. Kannon, M. L. Littman, and S. Vemphala, "Efficient Singular Value Decomposition via Improved Document Sampling," Department of Computer Science, Duke University, Durham, NC, Tech. Rep. CS-99-5, 1999.
- [6] C. R. Johnson, "A Gerschgorin-type lower bound for the smallest singular value," *Lin. Alg. Appl.*, Vol. 112, pp. 1–7, 1989.
- [7] T. Kato, "Upper and Lower Bounds of Eigenvalues," *Physical Review*, Vol. 77, pp. 413, 1949.
- [8] K. Kimura, M. Takata, M. Iwasaki, and Y. Nakamura, "Application of the Kato-Temple inequality for eigenvalues of symmetric matrices to numerical algorithms with shift for singular values," in *Proc. ICKS2008*, 2008, pp. 113–118.
- [9] K. Kimura, T. Yamashita, and Y. Nakamura, "On $O(N)$ formula for the diagonal elements of inverse powers of symmetric positive definite tridiagonal matrices," Department of Applied Mathematics and Physics, Kyoto University, Kyoto, Kyoto, Tech. Rep. 2008-010, 2008.
- [10] K. Kimura, T. Yamashita, and Y. Nakamura, "Conserved quantities of the discrete finite Toda equation and lower bounds of the minimal singular value of upper bidiagonal matrices," Department of Applied Mathematics and Physics, Kyoto University, Kyoto, Kyoto, Tech. Rep. 2011-003, 2011.
- [11] U. von Matt, "The orthogonal qd-algorithm," *SIAM J. Sci. Comput.*, Vol. 18, pp. 1163–1186, 1997.
- [12] B. N. Parlett and O. A. Marques, "An Implementation of the dqds Algorithm (Positive Case)," *Lin. Alg. Appl.*, Vol. 309, No. 1-3, pp. 217–259, 2000.
- [13] W. K. Pratt, *Digital image processing*, New York, USA: Wiley-Interscience Publishing, 1978.
- [14] M. Takata, K. Kimura, M. Iwasaki, and Y. Nakamura, "An Evaluation of Singular Value Computation by the Discrete Lotka-Volterra System," in *Proc. PDPTA2005*, 2005, Vol. II, pp. 410–416.
- [15] M. Takata, K. Kimura, M. Iwasaki, and Y. Nakamura, "Performance of a New Singular Value Decomposition Scheme for Large Scale matrices," in *Proc. PDCN2006*, 2006, pp. 304–309.
- [16] M. Takata, T. Konda, K. Kimura, and Y. Nakamura, "Verification of dLVv Transformation for Singular Vector Computation with High Accuracy," in *Proc. PDPTA2006*, 2006, Vol. II, pp. 881–887.

Evaluation of the SVM based Multi-Fonts Kanji Character Recognition Method for Early-Modern Japanese Printed Books

Manami Fukuo¹, Yurie Enomoto^{1†}, Naoko Yoshii¹,
Masami Takata¹, Tsukasa Kimesawa², and Kazuki Joe¹,

¹ Dept. of Advanced Information & Computer Sciences, Nara Women's University, Nara, Japan

² Digital Library Division, National Diet Library, Kyoto, Japan

Abstract - *The national diet library in Japan provides a web based digital archive for early-modern printed books by image. To make better use of the digital archive, the book images should be converted to text data. In this paper, we evaluate the SVM based multi-fonts Kanji character recognition method for early-modern Japanese printed books. Using several sets of Kanji characters clipped from different publishers' books, we obtain the recognition rate of more than 92% for 256 kinds of Kanji characters. It proves our recognition method, which uses the PDC (Peripheral Direction Contributivity) feature of given Kanji character images for learning and recognizing with an SVM, is effective for the recognition of multi-fonts Kanji character for early-modern Japanese printed books.*

Keywords: character recognition; SVM; digital archiving

1 Introduction

The national diet library (NDL) [1] in Japan keeps about 390,000 books dating from the Meiji and Taisyō era (1868-1926). The books cover a broad range including philosophies, literatures, histories, technologies, natural sciences, etc. Most of them are out of print and valuable materials in scholarly.

Generally, books in libraries read in hand have too many risks of aging or wearing, and loss by man-made source to open to the public. To solve the problem, the NDL started a project called "The Digital Library from the Meiji Era" [2,3]. In the project, early-modern printed books are recorded on microfiches page by page. The microfiches are converted into digital images and opened at the project Web site. Converting the books into digital images enabled the contents of the valuable books to be opened to the public while the original books are preserved in good condition. The NDL provides about 148,000 volumes with 101,000 titles in the digital library from their collection. Users can see the digital images of books whenever or wherever with the Internet connection.

The information including titles and author names of the books in the digital library is given as text data while main body is image data. There are no functions for generating text data from image data. Thus full-text search is not supported yet. To make early-modern printed and valuable books data more accessible, their main body should be given as text data, too. As described above, the number of the target books is so large that auto conversion is required.

If the conversion targets were general text images, they would have been converted into text data easily with some software of optical character recognition (OCR). However, most of early-modern printed books contain old Japanese characters that are not used now. Moreover, the fonts used in the early-modern printed books vary by publisher and year of publication. Existing OCR software cannot recognize right characters under these conditions.

To solve these problems, we proposed a multi-fonts Japanese character recognition method for early-modern printed books [4]. However, we selected just ten kinds of (Japanese) Kanji characters, which are commonly and frequently used, for the recognition experiment. Thus, the effectivity of the proposed method was not clearly shown. In this paper, we extend the kinds of Kanji characters for several recognition experiments to validate our method. To perform several recognition experiments, we need enormous character images with various font sets from early-modern books by various publishers. Therefore we need to perform automatic character clipping from images. Then, we extract the features of character images using a feature extraction method for handwritten Kanji character recognition because of the wide variety of font-sets and hard noises. Finally, the Support Vector Machine (SVM), which is one of promising recognition methods, is used for the recognition of the feature vectors.

The rest of this paper is organized as follows. In section II, we present the overview of our multi-fonts Kanji character recognition method. The evaluation method to validate the effectivity of our method is explained in section III. In section

[†] Currently works for Fujitsu Limited.

IV, we describe evaluation experiments and discuss the experimental results.

2 The Recognition Method

In this section, we give the overview of our multi-fonts Kanji character recognition method presented in [4].

For the character recognition of early-modern printed books, we think we should not use the typical OCR processes because each publisher might use different font sets in those days. The flow of our recognition method is shown below.

- a) An image containing a character is given as training data.
- b) Several pre-processes such as binarization, normalization and noise reduction are applied to the given image.
- c) The PDC (Peripheral Direction Contributivity) feature [5] is extracted from the pre-processed image, and each feature vector for the given image is labeled with the category according to the kind of the character included in the given image.
- d) A dictionary is generated by an SVM with the training data extracted in c). The dictionary is used for the recognition of test data.

Giving character images at a), several pre-processes are applied to each image. Any pixel in the images must be black or white when PDC features are extracted from the images. Thus binarization is applied to the images. In normalization, margins are removed from the images so that each character size is equivalent to be scaled. Noise reduction should be also applied since printing and archiving quality of early-modern printed books is mostly poor. Without noise reduction, noises of images would be recognized as character strokes to be extracted as unsuitable PDC features.

PDC feature vectors are calculated with category labels at c), and used for the learning phase of SVM at d). A training data for SVM learning is a set of PDC feature vectors and its label while a test data for SVM classification is a set of PDC feature vectors without label. All character images are converted to training or test data by processes a) to c). Half of the whole data are randomly selected from each category as training data, and the others are used for test data. An SVM learns separated hyper-planes in the PDC feature vector space with the training data. The trained SVM can classify test data according to the separated hyper-planes.

In our previous research [4], the SVM recognition experiments were performed for ten kinds of Kanji characters. Furthermore we compared the experimental result of SVM with a neural network (NN). The experimental result of SVM was the recognition rate of 97.8% while the NN was 77.6%. We confirmed that our SVM based method was more suitable for learning and classification of PDC features than the NN

based method. However, there are 6,349 kinds of Kanji characters, which Japanese Standards Association selects as the Japanese character code [6]. In addition, the structure of kanji characters is hierarchical: There are a lot of similar Kanji structures such as radical indices.

In this paper, we extend the kinds of Kanji character up to 256 kinds for several recognition experiments and validate the effectivity of our method. In [4], all character images for the recognition experiment were clipped from early-modern books by hand. Because of the large number of the kinds of Kanji characters for this experiment, we also implement automatic Kanji character clipping for early-modern books.

3 Automatic Kanji Character Clipping

The flow of automatic Kanji character clipping to extract each character image for the recognition experiments is shown below.

- 1) A page of the book images is given as an object image and divided into right and left parts.
- 2) Several pre-processes such as binarization, noise reduction and affine transform are applied to each divided half part of the object.
- 3) Layout analysis is applied to each part.
- 4) Character strokes are clipped within the layout analysis results.
- 5) Labeling is performed for each clipped character stroke so that each label is mapped to a character image.

Several pre-processes are executed on each part image. Binarization is applied to the image since the distribution of the vertically projected pixel values is used for extracting character domains. Noises of the image might be recognized as a false character domain. Then the noise reduction by a median filter is performed. Furthermore, distortion correction by an affine transformation is also applied since the distortion by capturing has a great influence to layout analysis. Therefore, layout analysis would be performed more correctly with noise reduction and distortion correction. The threshold is calculated with the distribution of the vertically projected pixel values, and layout analysis is executed. The auto-correlation function by Sondhi [7] is applied to the distribution of the vertically projected pixel values to calculate stroke spacing. Finally labeling is performed to each character stroke, which is clipped based on the stroke spacing. Each labeled part of connected black dots is recognized as a character domain to clip a character from the original book images.

In this paper, we use the character images clipped by the automatic character clipping. Figure 1 shows some examples

of clipped character images. Each character image is clipped from nine early-modern books with various publication ages and publishers so that we get character images with various font sets. It means we get nine kinds of PDC feature vector sets for each Kanji character. In this paper, a PDC feature vectors is calculated with nine character images in Fig. 1, for example. Furthermore, we calculate the averages of each dimension to obtain the standard deviation of the PDC feature vector. As the result, the standard deviation is 11.53, which seems to be an enough small value. That is, the fluctuation of characters are enough small. Therefore, we use nine character images clipped from different books for each character. Four character images for each character are randomly selected as training samples, and the others are used for test samples.

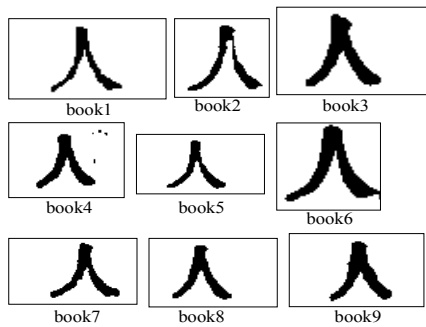


Figure 1 Clipped kanji character images from different books

4 Experiments

4.1 Experiment Method

We explain the data used in this experiment. To confirm the effectivity of our method, character images with various font sets should be collected from early-modern books of various ages. In the early-modern ages, different publishers used different font sets in general. Even the same publisher might use different font sets in different ages. Therefore, we need various character images for each kind of characters clipped from as many different font sets as possible. The question is how to get the sufficient number of samples for each target character. We use the Aozora Bunko [8] to check which books contain a target character.

The Aozora Bunko is a Web based online collection of the public domain literary works in Japanese. Users can download the works as text data. The number of works offered by the Aozora Bunko is about 7,200. All of them are also included in NDL's "the digital library from the Meiji era". The works in the Aozora Bunko can be used as a subset of the works in the digital library from the Meiji era. By using text data of the works in the Aozora Bunko, the character occurrence frequency can be calculated easily. The data set for the experiment is collected from text data in the Aozora Bunko, and image data corresponding to the data set are obtained from the digital library from the Meiji era as shown below.

Table 1 The list of nine early-modern works

Book Number	Title	Author	Publisher	Publication Year
1	L'Incident de Sakai	Mori Ougai	Suzuki Miekichi	1914
2	Like that	Mori Ougai	Momiyama bookstores	1914
3	The Boat on the Takase River	Mori Ougai	Shunyodo Publishing Co., Ltd.	1918
4	I Am a Cat	Natsume Soseki	OKURA Publishing Co., Ltd.	1905-1907
5	London Tower	Natsume Soseki	Senshokan	1915
6	Tobacco and the Devil	Akutagawa Ryunosuke	SHINCHOSHA Publishing Co., Ltd.	1922
7	Strange Reunion	Akutagawa Ryunosuke	KINSEIDO Publishing Co., Ltd.	1922
8	Returning a Favor	Akutagawa Ryunosuke	Jiritsusha	1923
9	One Day in the Life of Oishi Kuranosuke	Akutagawa Ryunosuke	Bungei shunju Ltd.	1926

Table 1 shows the list of nine early-modern works used for this experiment. Firstly, we examine all Kanji characters used in the nine early-modern works with the text data from the Aozora Bunko to find the Kanji characters that are used in the nine works in common. We find there are 262 kinds of commonly used Kanji characters. We randomly select several sets from 262 kinds of Kanji characters, and clip the Kanji character images from the nine book images of the digital library. In this experiment, we use the Kanji character sets of 16, 32, 64, 128, and 256 kinds to examine the recognition rate for each set.

The original book images are monochrome with 256 gray scales. Several pre-processes are applied to each Kanji character image. The image data are binarized in the first pre-process for PDC feature extraction. The second pre-process is noise reduction by a median filter. The median filter with a 3 by 3 mask is applied to the images. The normalization of size and position is performed by trimming the margins to clip the Kanji character image area. To fit the Kanji character images precisely into the square of 128 by 128 pixels, centering and affine transformation are applied. By these pre-processes, all Kanji character image data are converted to binary images with 128 by 128 pixels.

PDC feature vectors are generated from the pre-processed images and used for the learning phase of SVM. Five samples are randomly selected for each Kanji character and used as training samples. The other five samples are used for the recognition phase as test samples. We adopt an SVM as a recognition method.

4.2 Results

SVMs require careful parameter choices for their learning processes in general. In this experiment, we determined the parameters by grid-search and 5-fold cross-validation for the SVM. We use LIBSVM [9] for the experiments in this paper. Radial Basis Function (RBF) is used as the kernel function of the SVM. The numbers of Kanji characters sets are 16, 32, 64, 128, and 256, and each range of two RBF parameters, γ and C, is determined by grid-search. C is a penalty parameter for the SVM. Each element of PDC feature vectors is rescaled to the range of [-1, 1]. Each parameter for five experiments is shown as follows. In the cases of 16, 32, 64, 128, and 256, the parameters are set to $(\gamma, C) = (2^{-13}, 2^5), (2^{-9}, 2^3), (2^{-9}, 2^3), (2^{-11}, 2^5),$ and $(2^{-13}, 2^7)$, respectively. Note that these parameters are chosen just by a simple grid-search. Since the experiments in this paper are preliminary, we do not optimize the parameters.

Table 2 shows the experimental results. As is easily expected, the recognition rate becomes low as the number of Kanji characters increases. However, the recognition rate of each case always keeps more than 92%. Several examples of unrecognized samples are shown in Tab. 3, Tab. 4 and Tab. 5.

Table 2 The number of errors and the recognition rates

The number of Kanji characters	Correct/Test samples	Errors	Recognition rate[%]
16	64/64	0	100
32	124/128	4	96.875
64	248/256	8	96.875
128	477/512	35	93.164
256	949/1024	75	92.676

Table 3 Miss-recognized characters with the same radical indices

	Pre-processed image data	Correct character	Recognized character
Case1	違	違	遠
Case2	渡	渡	沈
Case3	感	感	思
Case4	聞	聞	間
Case5	過	過	通
Case6	連	連	遠
Case7	側	側	何
Case8	關	關	聞
Case9	問	問	間
Case10	後	後	微

4.3 Discussion

We analyzed the recognition results in detail to find three cases for miss-recognized Kanji characters. The three cases are listed below.

- The miss-recognized Kanji characters have the same radical indices.
- The miss-recognized Kanji characters do not have the same radical indices but similar structures.
- The miss-recognized Kanji characters do not have any similar structures.

Table 4 Miss-recognized characters with similar structures

	Pre-processed image data	Correct character	Recognized character
Case11	體	體	置
Case12	五	五	左
Case13	右	右	左
Case14	眞	眞	兵
Case15	場	場	現
Case16	床	床	成
Case17	申	申	出
Case18	時	時	持
Case19	變	變	幾
Case20	國	国	自
Case21	快	快	色
Case22	同	同	問
Case23	者	者	着
Case24	白	白	自

Table 5 Other miss-recognized characters

	Pre-processed image data	Correct character	Recognized character
Case25	歸	歸	深
Case26	餘	餘	深
Case27	空	空	草
Case28	張	張	紙
Case29	實	實	得
Case30	寢	寢	無
Case31	微	微	惡
Case32	微	深	抵
Case33	無	無	成
Case34	結	結	着

Table 3 shows some examples of miss-recognized Kanji characters with the same radical indices. Table 4 shows some examples of miss-recognized Kanji characters with similar structures. Recognition errors of cases 1-4 shown in Tab. 3 and cases 11-17 shown in Tab. 4 are caused by lack of character strokes removed at the pre-processes phase. Similar features of Kanji characters may be emphasized by lack of character strokes. In cases 1-4, similar features are radical indices. In cases 11-17, similar features are horizontal and vertical strokes and slant of character strokes. These false features may lead the SVM to miss-recognition.

In cases 5-8 and cases 18-20, the pre-processes seem to eliminate complex structures of the character strokes. Especially, cases 6, 8, 19 and 20 are much smeared images. The recognition error of case 9 is caused by noises in the image that are not removed at the pre-processes. In case 10 and cases 21-24, the possibility that the recognition errors are caused by noises is low since the pre-processed images have relatively few noises. In each occasion, correct cases and unrecognized cases have common features: Horizontal and vertical character strokes and their slants. These common features may lead the SVM to false recognition.

Table 5 shows some examples of miss-recognition among the Kanji characters that do not have any similar structures. The reason of miss-recognition in cases 25-28 is the same to cases 11-17. The reason of miss-recognition in cases 29-34 is the same to cases 5-8 and cases 18-20. Thus, noises prevent the original images from correctly clipping margins. Or they are recognized as some parts of character strokes. Therefore, the calculation of PDC feature vectors is greatly affected by noises in the original Kanji character images.

Table 6 Miss-recognitions by title

Book Number	Title	32 kinds	64 kinds	128 kinds	256 kinds
1	L'Incident de Sakai	0	0	0	1
2	Like that	1	1	7	12
3	The Boat on the Takase River	0	0	1	2
4	I Am a Cat	0	0	0	0
5	London Tower	2	1	6	10
6	Tobacco and the Devil	0	1	4	12
7	Strange Reunion	0	0	5	10
8	Returning a Favor	0	2	7	14
9	One Day in the Life of Oishi Kuranosuke	1	3	5	11

Table 6 reflects our discussion. It shows the number of miss-recognitions by title. Titles 2, 5, 7, 8 and 9 have a lot of miss-recognitions. Figure 2 shows examples of Kanji character images clipped from titles 2, 5, 7, 8 and 9. The Kanji character images in Fig. 2 are broken because of poor printing

and archiving quality. Furthermore, Tab. 7 shows the relation between the number of Kanji characters and the number of miss-recognitions for cases i), ii) and iii). The more the number of Kanji characters for experiments is increased, the more the ratio of cases i) and ii) to three cases. Therefore, it is observed that miss-recognitions caused by similar structures increase as the number of target Kanji characters increases in the experiments.

Therefore, the reasons of miss-recognition are considered as follows. Firstly, noises sometimes give bad effects to PDC feature extraction. The second reason is the similarity of Kanji characters: radical indices, horizontal and vertical character strokes, and their slants.

Table 7 Miss-recognitions by case

The number of Kanji characters	Case (i)	Case (ii)	Case (iii)
32	0	2	2
64	0	3	5
128	5	15	12
256	13	27	28

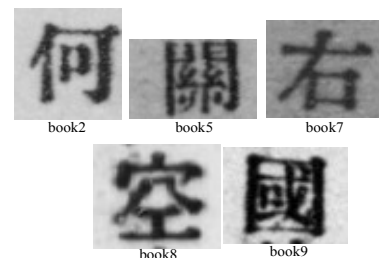


Figure 2 Exsamples of miss-recognized clipped Kanji characters

5 Conclusions

In this paper, we evaluated the SVM based multi-fonts Kanji character recognition method for early-modern Japanese printed books. To evaluate our recognition method, we used 262 kinds of Kanji characters, which are commonly used in nine early-modern titles from different publishers found in "The Digital Library from the Meiji Era". We applied automatic character clipping to the nine titles to clip Kanji character images. To extract features of Kanji characters, the PDC feature was calculated from the pre-

processes character images. For effective experiments, we generated five sets (16, 32, 64, 128, and 256) of Kanji characters. We selected each set from the 262 kinds of Kanji characters at random, and clip Kanji character images from nine different images of NDL's digital library. To recognize Kanji character images based on the extracted PDC features, their feature vectors were given to an SVM for learning. When the targets were 16, 32, 64, 128, and 256 kinds of Kanji characters, the recognition rates were 100%, 96.875%, 96.825%, 93.164%, and 92.676%, respectively. Each recognition rate for training samples was 100%.

We showed that our SVM based Kanji character recognition method could recognize printed Kanji characters clipped from early-modern Japanese printed books. However, two reasons of miss-recognition are considered. Firstly, miss-recognitions are caused by noises. Kanji character images clipped from early-modern printed books are usually broken or smeared because most of the early-modern printed books have been ill-preserved. The second reason is the similarity of Kanji characters: radical indices, horizontal and vertical character strokes, and their slants.

We think it will require some kind of hierarchical structures for learning data. In that case, the question is how 6,400 Japanese Kanji characters are divided in to a hierarchical structure, which is still an open problem. Another point of improvement is the pre-processes. We should improve the noise reduction to apply to heavy noises with poor printing quality on debased papers.

Acknowledgment

This work is partially supported by Grant-in-Aid for scientific research from the Ministry of Education, Culture, Sports, Science and Technology of Japan (MEXT).

References

- [1] National Diet Library: www.ndl.go.jp/en/index.html
- [2] Digital Library from the Meiji Era, kindai.ndl.go.jp/ (in Japanese)
- [3] Pamphlet of Digital Library from the Meiji Era, [kindai.ndl.go.jp/information/kindai\(eng\).pdf](http://kindai.ndl.go.jp/information/kindai(eng).pdf)
- [4] C. Ishikawa, N. Ashida, Y. Enomoto, M. Takata, T. Kimesawa, and K. Joe, "Recognition of Multi-Fonts Character in Early-Modern Printed Books," Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'09), 2009, pp.728-734.
- [5] N. Hagita, S. Naito and I. Masuda. "Handprinted Chinese Characters Recognition by Peripheral Direction Contributivity Feature", IEICE, Vol.J66-D, 10, pp.1185-1192, 1983. (in Japanese)
- [6] Japanese Standards Association: www.jsa.or.jp/default_english.asp
- [7] Man Mohan Sondhi: "New Methods of Pitch Extraction", IEEE Transactions on Audio and Electroacoustics, Vol.AU-16, No.2, June 1968, pp.262-266.
- [8] Aozora Bunko www.aozora.gr.jp/ (in Japanese)
- [9] V. Vapnik. "The Nature of Statistical Learning Theory". Springer-Verlag, 1995.

Optimization of the Particle-based Volume Rendering for GPUs with Hiding Data Transfer Latency

Kyoko Nakao, Erika Matsui¹, Naoko Yoshii, Masami Takata, Kazuki Joe

Graduate School of Humanities and Sciences,
Nara Women's University, Nara, 630-8506, JAPAN

Abstract – In this paper, we present the optimization of the particle-based volume rendering for GPU platforms. In general, data transfer between CPU and GPU accompanies long latency. Using page lock memory of the CUDA runtime API, data area is selected so that the data transfer between CPU and GPU becomes faster to reduce the execution time. In the meantime, using streams, the overlap of data transfer and the execution of kernels is achieved. As the result of experiment with the voxel data of a typhoon (1,188 × 979 × 64, 140MB), data transfer time and kernel execution time are improved and bring out about 30% performance of the GPU.

Keywords: GPGPU, volume rendering, optimization, HPC

1 Introduction

3D visualization technique is widely used for easy understanding of simulation and/or measurement results in the areas of science and technology, and for creating 2D images from CT and/or MRI data in medical fields. Simulation technologies are necessary to solve highly complicated problems. Recently it tends to require higher precision and resolution as the performance of computers increases. As the result, the volume data resulted from the simulation would enlarge and get complex. As a visualization method for such volume data, volume rendering techniques [1] are widely known to be effective. Volume rendering is used for visualizing not only the surfaces of 3D objects but also the inner information when 3D images are generated. Since it requires extremely data and computation intensive tasks, the computation time for visualization of huge data is enormously long. On the other hand, real time 3D visualization environments are preferable for efficient research activities where researchers explore new knowledge from data analysis with volume rendering.

In this paper, we present an optimization method for the particle-based volume rendering [2] using GPGPU. There are two features in the particle-based volume rendering. First, it does not need depth sorting of given sampling points by deleting implicit points with a z-buffer algorithm when

obscure and luminescent particles are projected to a screen. Therefore, it is suitable for the visualization of large-scale 3D data. The other feature is that the particle based model can display very fine objects such as fogs, smoke and clouds. GPUs are cost effective as well as a highly computational resource, and easy for any users to install. In this paper, we present an efficient application of the particle-based volume rendering to a GPU. As a development environment, we use CUDA [3], which is an integrated development environment of C language for GPUs by NVIDIA.

There are several papers that present improved implementations of the particle-based volume rendering for GPUs and/or CUDA. In [4] an acceleration method of the particle-based volume rendering for a GPU is reported, and the authors improved the method to a CUDA environment [5]. In [4,5], the authors present how fast the improved particle-based volume rendering is executed on a GPU and/or in a CUDA environment. So they just present “frame per second”. We are interested in the HPC aspect of volume rendering methods, especially the particle-based volume rendering. So we separate the particle-based volume rendering into HPC parts and drawing parts to present the improved performance in FLOPS.

The paper is constructed as follows. In section 2, we introduce the particle-based volume rendering. In section 3, we explain the optimization method in detail. In section 4, we validate the optimization method by experiments.

2 Particle-based Volume Rendering

In this section, we briefly introduce the particle-based volume rendering. Volume rendering is a technique to visualize data in 3D. It directly renders the data from their volume information. To generate 3D images, it is used for visualizing not only the surfaces of 3D objects but also the inner information. When we display surfaces of a 3D object by volume rendering, the volume information remains as well as the surface information. It has the advantage of applying a variety of visualization methods at any time because it always keeps the volume information. However, this process requires a huge amount of memory and computation resource because

¹ Currently working for Hitachi, Ltd.

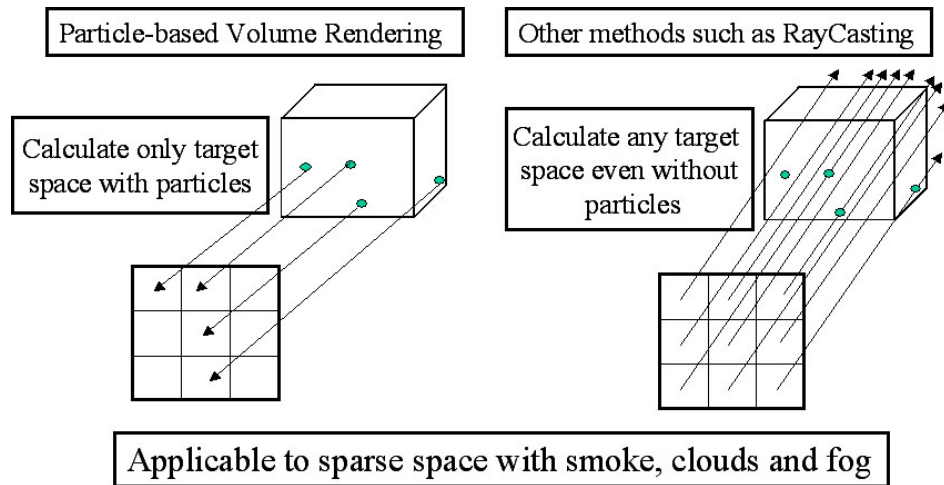


Fig. 1 The particle-based volume rendering and a volume rendering

of the large amount of volume information to be calculated. To reduce the amount of memory and computation, particle based volume rendering has been proposed [2].

Existing volume rendering algorithms capture sampling points along to or from the view direction so that the integration of view direction calculations are reused. Therefore, ordering sampling points is required. The ordering calculation requires large computation resource. On the other hand, the particle-based volume rendering generates particles based on particle density obtained from given volume data. In other words, the particle-based volume rendering generates luminescent particles with opacity property inside volume data to express the volume data as a discrete model using a set of luminescent particles with opacity. The model does not require sorting sample points toward the view direction, so the calculations can be performed in any order. Therefore, it is one of the most effective visualization methods for large volume data. During the particle-based volume rendering, the order to project particles to a screen is free, and the transparency of particles is very simple because hidden point removal operations are executable just by the comparison of the depth of each particle. Also, when there is no data in the target space, no calculation is performed because the view directions from particles are toward the screen. Thus, it is suitable for sparse volume data. Figure 1 shows the calculation difference between an existing volume rendering and the particle-based volume rendering. The particle-based volume rendering algorithm is given below.

- Step (1) Load voxel data
- Step (2) Calculate the opacity by a transfer function
- Step (3) Calculate the particle density by the opacity
- Step (4) Calculate the number of particles generated from the particle density
- Step (5) Calculate colors and positions of the generated particles
- Step (6) Project them on a screen

- Step (7) Sub-pixel processing

First, voxel data are loaded in Step (1). The voxel data are converted into available scalar values because they are stored in a binary format. In step (2), the data are sent to a transfer function so that the opacity of each voxel is calculated. In step (3), the particle density for each voxel is calculated by the opacity. In step (4), the number of particles to be generated for a volume is determined from the particle density. In step (5), the generated particles are located in a space for the volume by random, and their colors are determined in reference to the voxel data. In step (6), the generated particles get perspective projection and the screen color is determined. In step (7), the colors of the pixels on the screen image are averaged and alpha blended so that finer drawing is obtained.

3 Optimization

In this section, we propose an optimization method for the particle-based volume rendering using GPGPU. In subsection 3.1, we explain the target platform. In subsection 3.2, we explain data transfer between the CPU and the GPU of the target platform. In subsection 3.3, we propose an optimization method for the particle-based volume rendering.

3.1 Target platform

Table 1 shows the overall performances of Tesla C1060 and GeForce8400GS that are included in the target platform. Tesla C1060 is the first chip for GPGPU provided by NVIDIA. The performance of Tesla's floating-point operations is extremely high compared with existing GPUs, and used mainly for large-scale simulation and of high quality image generation. It is intended for the high performance computing market. However, we cannot use C1060 to render with OpenGL because it is not equipped with video output. As we explain above, the purpose of this paper is to

demonstrate the performance of the particle-based volume rendering with GPGPU. So we use GeForce8400GS to render with OpenGL. GeForce8400GS is a low-end model of GeForce 8 series with a memory system compatible to DDR2.

Tab. 1 Specification of C1060 and 8400GS

	Tesla C1060	GeForce 8400GS
Peak Performance (Gflops)	933	67
# of SPs	240	16
Clock (MHz)	1300	900
Memory Bandwidth (GB/sec)	102	6.4
Memory Size (MB)	4096	512
Memory Clock (MHz)	1600	800

3.2 Data transfer between CPU and GPU

The data transfer latency between the host PC and GPUs via PCI-e is considerable. To reduce the latency, we use page-locked memory provided by CUDA runtime APIs to allocate memory blocks for fast data transfer. A CUDA-enabled GPU can perform the execution of a CUDA kernel function and the communication between the GPU and the host PC simultaneously. The condition for the simultaneous execution is that there is no data dependency between the kernel function and the data transfer. To keep the condition, we use streams to manage it by stream number. A series of processes given the same stream number is considered to have data dependency. In addition, a stream is available only for the data in page-locked memory.

3.3 Optimized procedure

For the algorithm of the particle-based volume rendering presented in subsection 2.1, we apply some optimizations for

GPGPU. Figure 2 shows the optimized procedure. Since the voxel data used in the experiments (section 4) is stored in a three dimension array (1,188 by 979 by 64), we regard blocks as two dimension data and threads as vector data. The optimized procedure is enumerated below.

- (1) Read voxel data
- (2) Send voxel data to Tesla C1060
- (3) Calculate the opacity and the density of particles
- (4) Calculate the number of generated particles
- (5) Send the number of generated particles to the host
- (6) Select the storage location for generated particles
- (7) Send the storage location for generated particles to Tesla C1060
- (8) Calculate the color, the location, and the gradient of particles
- (9) Send the results to the host
- (10) Send the particle data to GeForce 8400GS
- (11) Perspective projection of the particle data
- (12) Display the calculation results to a screen

In (1), a voxel data set is given to the host PC and the voxel data is sent from the host PC to Tesla C1060 in (2). In (3), the opacity and the density of particles are calculated. These operations are performed in parallel for each voxel data as shown in Fig. 3. The number of generated particles is calculated in (4), and positions, colors and gradients of generated particles are calculated in (8), as shown in Fig. 4, in parallel for each sub-voxel data. In (5), the number of the generated particles is send back to the host. In (6), the locations to store the generated particles are calculated. This calculation has to be performed in the host because it requires data exchange between blocks in the GPU. The locations to store the generated particles are sent to Tesla C1060 in (7). Since Tesla C1060 does not have rendering operations, the particle data is sent to GeForce 8400GS via the host in (9) and (10). In (11) and (12), the particle data gets perspective

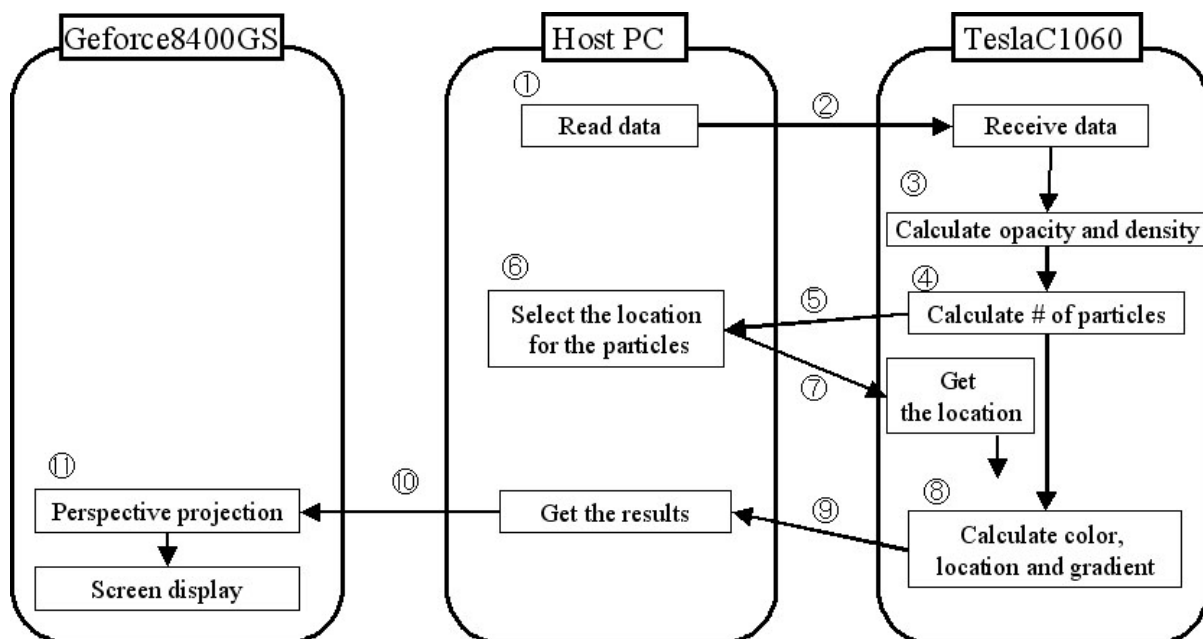


Fig. 2 Optimized procedure

projection to display on a screen using OpenGL functions.

In (2), (5), (7), (9), and (10), data transfers between GPUs and the host are required. To reduce the communication latency, we use the page-locked memory for particle data, storing locations of generated particles, particle colors, particle positions, and an array for gradient values. In addition, multiple streams are used for the data transfers: (2) and (3) are operated with stream 1, and (7), (8), and (9) are with stream 2. Memory references in CUDA make use of grid blocks. Using the grid blocks, block IDs and thread IDs are obtained, and the IDs are used for memory reference to perform parallel processing. Equation (1) is used for calculating the position of the volume data to be accessed.

$$index = (gridDim.x * blockDim.x * blockIdx.y) + (blockDim.x * blockIdx.x) + threadIdx.x \quad (1)$$

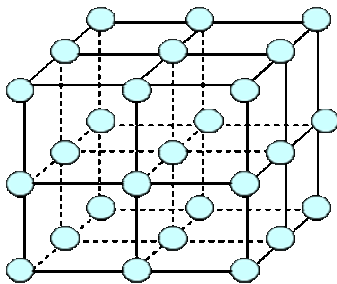


Fig. 3 Parallel calculation by voxel

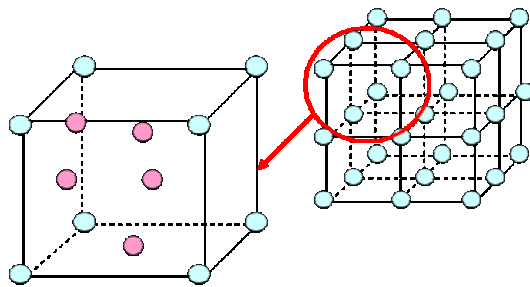


Fig. 4 Parallel calculation by sub-voxel

4 Experiments

To validate the performance of our optimization method for the particle-based volume rendering, we performed some experiments. The experiment environment is as follows. The host PC consists of Intel (R) Core (TM) i5 760@2.80GHz and 4GB Memory with CUDA2.3. The GPUs are GeForce 8400GS and Tesla C1060. We use a voxel data set of typhoon density of which size is 1,188 by 979 by 64 (74,435,328) for the 140MB.

For Tesla C1060, we perform experiments to validate the optimization effect for the calculations of the number of the generated particles and particle density, opacity, position, color and gradient of the generated particles. We also perform the same experiments by the host for the comparison purpose. The parameters for the particle-based volume rendering are

$TF(S) = S/19840$, $\alpha = TF$, where TF stands for Transfer Function and S is a scalar value. In subsection 4.1, we compare the execution times of Tesla C1060 with and without streams. In subsection 4.2, we compare the performance difference between the host and Tesla C1060.

4.1 Comparison of executions with and without streams

In this subsection, we investigate the effects of stream. As described in subsection 3.3, we use two streams. Stream 1 includes the density calculations for particles as kernel and the voxel exchanges as data transfer. Stream 2 includes the the position, color and gradient calculations for particles as kernel, the location to be stored, color, position and gradient exchanges as data transfer.

Tab. 2 Execution time (msec) with/without streams

	without stream	with stream
kernel (particle density)	0.066	0.025
kernel (position, color, gradient)	0.013	0.013
data transfer (voxel data)	24.646	0.044
data transfer (location of particles)	48.450	0.005
data transfer (position, color, gradient)	96.741	0.008

Table 2 is the execution times of each kernel and data transfer with and without streams. As for the kernel execution times, the calculation of particle density with stream 1 is faster than without stream. This is because there are overlays between calculation and data transfer in stream 1 (kernel execution for particle density). On the other hand, the kernel with stream 2 (position, color and gradient) does not get any performance benefit. This means that the kernel with stream 2 does not contain any data transfer.

As for the data transfer executions, the execution times of voxel data, locations to be stored for particles, and position/color/gradient of particles with streams are faster than without streams 560, 9,690, and 12,729 times, respectively. The reason of the performance gain is considerable overlays between calculation and data transfer as well as low-latency of page-locked memory. Thus, the communication time is hidden by the use of stream and page-lock memory.

4.2 Comparison of execution times of Host and C1060

In this subsection, we compare the computation times of particle density, the number of generated particle, and their positions/color/gradient between the host (a 2.8GHz CPU)

and Tesla C1060. The computations are kernels with stream 1 and stream 2 described in subsection 3.3. Table 3 shows the execution time of each kernel.

Tab. 3 Performance comparison (msec)

	Host	C1060
particle density	5.396	0.025
# of generated particles	4.292	0.015
position, color, gradient	4.744	0.013

We find that Tesla C1060 is 215, 286, and 364 times faster than the 2.8GHz CPU for the calculation of particle density, the number of generated particle, and the position/color/gradient of particles, respectively. The performance gain comes from parallel executions of Tesla C1060 kernels and hiding the communication latency by overlaying data transfer and kernel executions as described in subsection 4.1.

	# of operations	performance (Gflops)	peak ratio
particle density	3,126,283,776	125.05	0.13
# of generated particles	2,486,617,812	165.77	0.18
position, color, gradient	3,903,282,340	300.25	0.32

Tab. 4 Performance of C1060 kernels

Table 4 shows the number of operations, the performance of each C1060 kernel in Giga FLOPS and their peak performance ratios. We observe that the kernel for calculating (position, color, gradient) has achieved 300GFLOPS, which is almost 1/3 of Tesla C1060's peak performance. Even the worst performance of the kernel for (particle density) is 125GFLOPS, which is considered as highly optimization.

In this experiment, we confirmed that our optimization of the particle-based volume rendering for Tesla C1060 recorded enough computing performance as GPGPU. However, we have not implemented the effective use of shared memory of Tesla C1060 for our method. We expect that we can improve the lower performance kernels by using the shared memory.

5 Conclusions

In this paper, we took note of the particle-based volume rendering suitable for visualization of very large volume data to speed up its main part using GPGPU. As optimization methods, we calculated the opacity and the density of each particle in parallel by voxel, and computed the number of generated particles, the position, the color and the gradient of the generated particle in parallel by volume. Since the data transfer latency between the host and Tesla C1060 is considerably long, we used page-locked memory of CUDA runtime API to allocate the array data for generated particles, their storing positions, and their color/position/gradient so that the data transfers and the kernels were executed with two streams simultaneously. A stream contains the kernel (particle density) and the data transfer (voxel data) while the other stream contains the kernel (position, color, gradient) and the data transfer (storing location, position, color, gradient). As for the data transfer, the transfer times of voxel data, storing locations, and position/color/gradient of particles with streams are faster than without streams 560, 9,690, and 12,729 times, respectively. The calculation times of Tesla C1060 for particle density, the number of generated particle, and the position/color/gradient of particles is 215, 286, and 364 times faster than the 2.8GHz CPU, respectively.

In the experiments, we found that a kernel had achieved 300GFLOPS performance, which is almost 1/3 of Tesla C1060's theoretical performance. So far, there are several reports about the performance of the particle-based volume rendering using GPU, but most of them are interested in the drawing performance. In this paper, we showed that the optimized implementation of the particle-based volume rendering achieved 1/3 performance of Tesla C1060's peak. This is meaningful since it is reported from the HPC site using GPGPU.

We are working for design a new architecture with high-end GPGPU boards and several normal GPU for remote sites. The results of this paper (Tesla C1060 as a high-end GPGPU board and GeForce 8400GS for remote) are to be used for the new architecture design, and it is our future work.

Acknowledgement

We would like to thank Prof. N. Sakamoto and Prof. K. Koyamada of Kyoto University for their helpful advices.

References

- [1] R. A. Drebin, L. Carpenter, and P. Hanrahan : Volume rendering. In Computer Graphics (Proceedings of SIGGRAPH '88), Volume 22, pp. 65–74 (1988).

[2] N. Sakamoto, J. Nonaka, K. Koyamada, S. Tanaka: "Particle-based Volume Rendering", Asia-Pacific Symposium on Visualization (APVIS 2007), pp.129-132 (2007).

[3] <http://www.nvidia.com/object/cuda_home.html>.

[4] D. Zhongming, T. Kawamura, Naohisa Sakamoto, Koji Koyamada : "GPU Acceleration of Improved Particle-based Volume Rendering for Irregular-grid Data", Proceedings of International Conference on System Simulation and Scientific Computing 2008, pp.685-692 (2008).

[5] Z. Ding, T. Kawamura, N. Sakamoto, K. Koyamada : "Particle-based Multiple Irregular Volume Rendering on CUDA", Simulation Modelling Practice and Theory, 18, 1172-1183 (2010).

A Real-time Analysis Environment for a Wireless BMI Device Enobio

Yu Ishikawa, Sanae Teramae¹, Naoko Yoshii, Masami Takata, Kazuki Joe
Graduate School of Humanities and Sciences,
Nara Women's University, Nara, 630-8506, JAPAN

Abstract – In this paper, we present a real-time analysis environment for an electroencephalogram system Enobio. The analysis environment consists of data acquisition, data processing, data display and GUI modules. To validate the real-time performance of the analysis environment, several experiments are performed. We investigate the difference among various algorithms and/or libraries of DFT and ICA so that we select the fastest data processing. Using the selected data processing functions, we show the delay time of the analysis environment from data acquisition to data display is almost one second.

Keywords: BMI, real-time data analysis, EEG, Enobio

1 Introduction

Brain-Machine Interfaces (BMIs) [1] are a technology to explore thinking states of human beings with biological information taken from brain for manipulating various machines just by their thinking, and are recently receiving increasing attention. The biological information is given as electroencephalogram (EEG) and there are many measurement technologies developed for EEG. The existing BMIs are roughly divided in two groups: large-size devices having multichannels to measure and analyze EEG for various purposes with high accuracy, and small-size devices having a single channel to measure a part of EEG for limited purposes with low accuracy. No need to say, the former is expensive and for experts while the latter is cheap and for toys. We are interested in developing a BMI with multichannel and a small-size to be used in daily life. Enobio [2] is a wearable, modular and wireless electrophysiology sensor system, and we use Enobio for our BMI. In this paper, we present a real-time multichannel EEG analysis environment using Enobio.

The analysis environment presented in this paper consists of four modules: 1) the data acquisition module to get stream data from Enobio via TCP/IP, 2) the data processing module to apply ICA (Independent Component Analysis) [3] and DFT (Discrete Fourier Transform) to obtained data for EEG analysis preprocesses, 3) the data display module to display

the stream data and/or preprocessed data using OpenGL [4], and 4) the GUI module.

The paper is constructed as follows. In section 2, we give an overview of Enobio. In section 3, we present a real-time analysis environment using Enobio. In section 4, we evaluate the environment by experiments.

2 Enobio

Enobio [1] developed by Starlab is a wireless electrophysiology recording system for EEG. It has four electrodes and is classified into scalp electrodes and bipolar derivation. A BMI using Enobio is classified into output and non-invasive systems [5]. Figure 1 shows an Enobio, and the specification of Enobio is shown in Tab. 1. Enobio communicator is lightweight (65g) and has a wireless signal transfer mechanism, so it is possible to measure EEG of which targets move relatively freely. Enobio is used for EEG, Electro-oculogram (EOG) and Electro-cardiogram (ECG). Among the supported functions, Enobio supports TCP/IP just for EEG, so we deal with EEG in this paper. The stream data of Enobio range from 0 to 65535 in micro volt. The real-time output data of four channels via TCP/IP are shown as follows. The first and the second bytes of the output data present the MSB and the LSB bytes of the first value of channel 1, respectively. The third and the fourth present the second value of channel 1, and so on.



Figure 1 Enobio

¹ Currently works for Mitsubishi Electric Corporation.

Table 1 Specification of Enobio

Number of Channels	4
Input	EEG, EOG, ECG
Output	Digital streaming over TCP/IP
	Raw data files – ASCII output and EDF compliant
Sampling	16 bit sampling at 250 S/s
System Requirements	Windows XP/Vista(32bits) or MAX OS X (Intel)
	1GB RAM or more
	JAVA 1.6 RE for Windows and JAVA 1.5 RE for MAC OS X
	USB2.0 port
	CPU Pentium D or better
Dimensions	660x550x250 mm
Weight	65 g



Figure 2 Enobio's optional electrodes

Since each channel data by clock consists of two bytes, the following calculation is required to get the actual electric potential.

$$1st\ value\ of\ channel\ 1 = 256 \times byte1 + byte2$$

$$1st\ value\ of\ channel\ 2 = 256 \times byte3 + byte4$$

$$1st\ value\ of\ channel\ 3 = 256 \times byte5 + byte6$$

$$1st\ value\ of\ channel\ 4 = 256 \times byte7 + byte8$$

The drawback of Enobio was its limitation of sensor position: just four positions around frontal cortex. Since our BMI system is to be used for exploring EEGs in daily life to support the person wearing the BMI system, we needed more flexibility about the kind of EEG data, namely sensor positions.

Recently Starlab announced that they provide optional electrodes, which can be located with a special hat. The Enobio user can replace the existing four electrodes with the optional electrodes to get other EEGs. Figure 2 shows the optional electrodes.

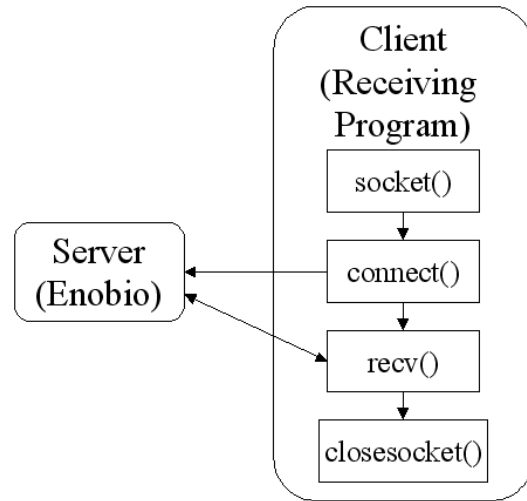


Figure 3 Processing flow between server and client by winsock

3 Real-time Enobio Analysis Environment

In this paper, we use a small-size, lightweight and multichannel EEG measurement system Enobio. Although Enobio provides attachment applications, it is insufficient for our target BMI development to observe and analyze the Enobio stream data. Thus, we present a real-time analysis environment for multichannel EEG Enobio to take advantage of its TCP/IP output function. We use Windows XP as the development environment.

3.1 The data acquisition module

Enobio supports streaming output via TCP/IP for measured data. We use winsock for real-time data reception functions via TCP/IP in our developing system. Figure 3 shows the processing flow between Enobio at server-side and the receiving program at client-side. First, the receiving program creates a socket by a socket function. Next, a TCP/IP address is specified for the connection by a connect function. The target address is obtained by the pre-defined number by an attachment application of Enobio. Then, the receiving program receives the data by the recv function with the necessary number of bytes and the storage location.

3.2 The data processing module

In this subsection, we expound some signal processing methods applied to the obtained data in the data processing module. In 3.2.1, we describe ICA (Independent Component Analysis) to separate independent components from the stream data and to remove noises. In 3.2.2, we describe DFT for the frequency distribution analysis of the stream data.

3.2.1 ICA

We aim to develop a BMI used in daily life, so we do not suppose any BMI measurement under special conditions

such as shield rooms to block electromagnetic waves from various peripheral devices and limitation of targets activities where they are forced to keep quiet or close eyes. Therefore, the stream data out of Enobio may include noises from peripheral power supplies and from eye movements or blinks of the targets. The received signals from an electrode can be mixed with other independent EEGs from multiple locations in a brain caused by electrodes arrangement. So we apply ICA to the stream data to remove noises and extract independent components as a pre-process for our BMI. In this paper, we adopt FastICA [6], which is known to have the fast convergence ability, as an ICA algorithm. We use IT++ library that includes the FastICA implemented in C++. The FastICA class library in IT++ library provides separate operations by selecting the type of orthogonalization or nonlinear function methods with an updating rule for the reconstruction matrix W .

3.2.2 DFT

In EEGs, the frequency components change by cognitive activities such as visual and auditory sense. We believe that human thinking states can be detected by observing the change of frequency components of EEGs to capture the features. Thus, we apply DFT (Discrete Fourier Transform) to observe the frequency changes of EEGs from Enobio. DFT of n discrete points x_i ($i = 0, 1, \dots, N - 1$) is defined by expression 1.

$$X_k = \sum_{i=0}^{N-1} x_i e^{-j \frac{2\pi ki}{N}} \quad (k = 0, 1, \dots, N - 1) \quad (1)$$

Let x_i be a complex signal where $x_i = Re(x_i) + jIm(x_i)$ and resolve expression 1 into the real part and the imaginary part by Euler's formula $e^{i\theta} = \cos \theta + i \sin \theta$. We express the real part of X_k is $Re(X_k)$ and the imaginary part is $Im(X_k)$. The stream data in this paper are presented as real, so we just treat the real part $Re(X_k)$ as follows.

$$Re(X_k) = \sum_{i=0}^{N-1} \{x_i \cos(2\pi ki/N)\}$$

$$Im(X_k) = \sum_{i=0}^{N-1} \{-x_i \sin(2\pi ki/N)\}$$

The frequency component size is represented below.

$$|X_k| = \sqrt{Re(X_k)^2 + Im(X_k)^2}$$

We apply the above expressions to the stream data (250 samples per second) from Enobio for observing the frequency distribution.

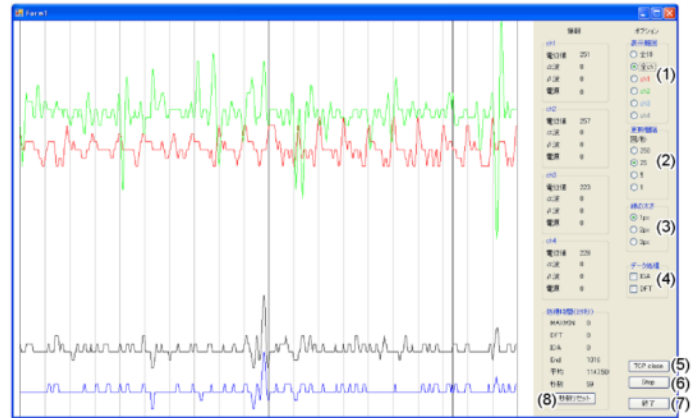


Figure 4 GUI

3.3 The data display module

Using OpenGL, the data display module visualizes the obtained data in the data acquisition module and/or the data processing module. The drawing target data are the original stream data, the DFT applied results, the ICA applied results and the DFT results after ICA. They are drawn in a form of polygonal line graphs. The drawing area has the size of 700 by 900 pixels for presenting last two seconds information of each channel, where the vertical and horizontal axis represents electric potential and time, respectively. The attachment application of Enobio shows each channel information on different graphs. This display method is suitable for observing potential changes by channel, but it is difficult to find potential relation among four channels. The display data of the attachment application uses only the original stream data, so it cannot show any pre-processed data such as ICA and/or DFT. Thus, for our developing analysis environment, we implement the following functions which are not included in the Enobio attachment application.

The total window to display the whole data

A graph window is used for displaying four channels data simultaneously. Herewith, we get the information about relative potential difference among four channels. Furthermore, each channel is uniquely colored for easy understanding of the potential differences.

Zooming a single channel

The data in a single channel can be zoomed into the whole screen. We can observe more detailed wave forms than the original graph of Enobio.

Grid

We improve observational efficiency with drawing separator lines, where the vertical and horizontal axis represents electric potential and time, respectively. The horizontal lines are drawn by 10,000 microvolts and the vertical lines by 100 msec, which construct grids on the graph.

Table 2 Average processing time in the data processing module

	ICA								DFT
	DEDL				SYMM				
	POW3	TANH	GAUSS	SKEW	POW3	TANH	GAUSS	SKEW	
data 1	21	2103	3220	3435	166	19	14	12	15
data 2	2931	14276	3500	21003	8	18	24	20	16
data 3	6656	6192	2709	105375	15	29	30	11	15

Line width

The line width of the grids is changeable. We get detailed wave forms when the line width is fine while it is easy to observe enhanced wave forms when the line width is thick.

Drawing intervals

We can change the interval between drawing refreshment. Compared to the data receiving time, drawing a wave form needs longer time. To keep real-time performance in observing wave forms, we reduce the number of drawing refreshments per second.

3.4 GUI

We develop a GUI to change drawing methods in the data display module and to control pre-processes of the stream data. Figure 4 shows the developed GUI.

The GUI is constructed with three parts: the graph display area, the information display area and the control area. The graph display area is used for displaying the graphs drawn in the data display module. In the information display area, the raw stream data and/or processing time are displayed by integer. The control area includes several buttons to control each component part of the analysis environment. The following eight kinds of buttons are located in the graph display area and the information display area.

- (1) Display range
- (2) Refresh interval
- (3) Line width
- (4) Data processing
- (5) TCP open/close
- (6) Start/Stop
- (7) Exit
- (8) Reset time counter

Button 1 defines the display range of a wave form. In the total window for the whole data, the vertical axis drawing range is limited between 0 to 65535 microvolts in the data display module. Ch-all represents all channels and "ch-1" to "ch-4" represents each channel, where calculation results from the minimum potential to the maximum potential are used for the display range of the vertical axis. Button 2 defines the refreshment rate for drawing the wave form in the data display

module. Button 3 changes the line width of wave forms. Button 4 controls the pre-processes for the stream data. The button 4 is used for selecting "no pre-process", "just DFT", "just ICA", and "DFT right after ICA". Button 5 gives TCP/IP client services. The button is the toggle for "TCP open" and "TCP close" which creates a socket in the data acquisition module and closes the process, respectively. Button 6 is used for starting or stopping the acquisition of the stream data. By button 7, the program is terminated. Button 8 resets the time counter. The time counter increases by one each time when 250 data are drawn, and is used for time keeping by second.

4 Experiments

In this section, we validate the real-time performance of the developed analysis environment by some experiments. First, we measure each processing time of ICA and DFT in the data processing. To determine which kind of orthogonalization and nonlinear functions are effectively performed in the data processing module, we investigate the processing time for several evaluation functions and optimization methods of ICA in the experiment. Then, we measure the elapsed time from the acquisition of output data from Enobio in a second to the completion of drawing a wave form in the experiment. Furthermore, we compare the time differences among update intervals for drawing with or without data processing. This experiment is performed on a computer with an Intel(R) CoreTM2 Duo E8400 3GHz CPU and 3GB memory.

4.1 Experiments in the data processing module

We measure the processing time of the data processing module, which is estimated as the most computation intensive in the proposed analysis environment. In this experiment, we measure the application of ICA and/or DFT to 250 samples from each Enobio channel by second. We adopt FastICA as an ICA algorithm, and use the IT++ library where FastICA is implemented. When the FastICA of IT++ library is executed, the types of orthogonalization and nonlinear functions are selected by option. In the FastICA class library, orthogonalization is selectable from Gram-Schmidt orthogonalization (DEFL) and symmetric orthogonalization (SYMM), while the nonlinear function $g(y)$ is from y^3 (POW3), $\tanh(y)$ (TANH), $y \exp(-y^2/2)$ (GAUSS) and y^2

Table 3: Average processing time under various conditions

Update Intervals \ Processes	250	125	50	25	1	0
Non	1400	1050	1006	1007	1006	1007
DFT	1409	1060	1007	1007	1007	-
ICA	1409	1059	1008	1006	1006	-
DFT&ICA	1422	1070	1007	1007	1007	-

(SKEW). We measure each processing time for all the combinations in this experiment.

Table 2 shows the average processing times of the application of ICA and DFT to three kinds of EEG data out of 100 trials. The data used in the experiment is from a pre-stored text file generated by Enobio. The table indicates that SYMM is faster than DEFL. SYMM calculates all independent components at once, while DEFL calculates each independent component recursively one by one. This is the reason of performance difference. Among SYMM, the processing times with POW3 and SKEW options are relatively short. However, the POW3 option causes performance variation depends on data.

From the above considerations, we adopt SYMM and SKEW with the shortest processing time as well as enough stability for FastICA options to observe wave forms in real time. Next, we discuss DFT for observing the frequency components in EEG data. The processing time of DFT is about 15 msec for any chunk of EEG data stream, and it is enough short compared with one second of each chunk. Thus, it keeps the analysis environment with real-time performance to implement DFT.

4.2 Experiments for Drawing Intervals

We measure processing time for 250 data from each channel of Enobio starting from the data acquisition until the completion of drawing wave forms to validate the real time performance of our analysis environment for Enobio. In the experiment, we investigate several measurement conditions of various drawing intervals and pre-processes presence. The conditions for drawing intervals are 250, 125, 50, 25, 1 and 0 times in a second and "no pre-process", "just DFT", "just ICA", and "DFT right after ICA" for pre-processes presence. Drawing interval 0 means just the processing acquisition for stream data without calling the data display module and it does not use any pre-process.

Table 3 shows the average of 100 measurements for the processing time under various conditions. It shows that the processing time is over 1,000 msec under any condition, even with drawing interval 0, which means just data acquisition time. Drawing intervals of 50, 25 and 1 take the processing times as long as drawing interval 0 without pre-processes. In the meantime, drawing intervals of 250 and 125 take longer processing times than the other conditions and pre-processes.

It is easily expected that the processing times depend on the cumulative waiting time for data reception from Enobio. The data sampling rate of Enobio is 250 Hz so that a chunk of stream data is sent from Enobio every four msec. Therefore, the data acquisition module has a waiting time from a data reception to the next data reception. In the case of drawing intervals of 50 or fewer times, since the sum of the total drawing time and the processing time is shorter than the cumulative waiting time, it is almost same to the case of drawing interval 0 that is just the data acquisition time and waiting time. In the case of drawing interval of 250 and 125, the sum of the total drawing time and the processing time is longer than the cumulative waiting time, which means that the drawing module cannot draw within a waiting time. In summary, the real-time performance of our analysis environment is kept when the drawing intervals are less than equal 50.

5 Conclusions

In this paper, we presented a real-time multichannel EEG analysis environment to support BMI research using Enobio which is a wearable, modular and wireless electrophysiology sensor system. The proposed environment consists of four modules: the data acquisition module, the data processing module, the data display module, and the GUI module. The data acquisition module receives stream data from Enobio via winsock. In the data processing module ICA and DFT are implemented for EEG observation analysis. The data display module is to draw wave forms of the stream data with OpenGL, and several functions which are not included in the Enobio attachment application have been implemented. In the GUI module, many functions for starting and stopping the data acquisition, controlling pre-processes for the stream data, and setting drawing wave forms have been implemented with the Windows form.

To validate the real-time performance of the analysis environment, we performed some experiments to measure processing times. First, we measured processing times of the data processing module where ICA and DFT are implemented. The experimental results showed that the processing time by the combination of SYMM and SKEW is the shortest among FastICA library options used for the ICA implementation. Next, we measured various processing times from the data acquisition to the wave forms drawing with changing processes applied to the stream data and drawing intervals of the analysis environment. The experimental results showed that we need the drawing intervals of 50 or fewer to the keep real-time performance of the developed analysis environment.

Acknowledgement

We would like to thank Dr. Masayo Haneda of Starlab for her helpful comments.

References

- [1] Guido Dornhege, José del R. Millán, Thilo Hinterberger, Dennis J. McFarland and Klaus-Robert Müller; *Toward Brain-Computer Interfacing*, MIT Press (2007).
- [2] <http://starlab.es/products/enobio>
- [3] Aapo Hyvärinen, Juha Karhunen, Erkki Oja : *Independent Component Analysis*, Wiley-Interscience (2001).
- [4] <http://www.opengl.org/>
- [5] Mikhail A. Lebedev and Miguel A.L. Nicolelis: *Brain-machine interfaces: past, present and future*, *TRENDS in Neurosciences*, Vol.29, No.9, pp.536-546 (2006).
- [6] A. Hyvarinen and E. Oja.: *A Fast Fixed-Point Algorithm for Independent Component Analysis*, *Neural Computation*, 9(7), pp.1483-1492 (1997).

Distributed PACS using Network Shared File System

Tomoyuki Hiroyasu
Faculty of Life and Medical Sciences
Doshisha University
Kyoto, Japan
tomo@is.doshisha.ac.jp

Yoshiyuki Minamitani
Graduate School of Engineering
Doshisha University
Kyoto, Japan
yminamitani@mikilab.doshisha.ac.jp

Masato Yoshimi
Department of Science and Engineering
Doshisha University
Kyoto, Japan
myoshimi@mail.doshisha.ac.jp

Mitsunori Miki
Department of Science and Engineering
Doshisha University
Kyoto, Japan
mmiki@mail.doshisha.ac.jp

Abstract—In this paper, a distributed PACS (Picture Archiving and Communication Systems) using a network shared file system is proposed. Several distributed hospitals and sites use PACS to manage medical images. For these distributed sites to work together, it is necessary to construct integrated PACS for these distributed files. In the proposed system, the DICOM (Digital Imaging and Communications in Medicine) metadata and the DICOM image data are registered separately on the server. In the proposed system, several operations can be performed using only the metadata. For example, images can be searched using only the information stored in the metadata, and therefore the client can find the target image without downloading the whole DICOM data. DICOM image size is usually huge. Therefore, if a file can be found or transformed without downloading the whole data, the file operation can be performed very quickly. To implement the proposed system, the open source Gfarm network shared file system, which treats metadata as XML data, was used. This paper describes construction of the proposed system, and discusses experimental comparison of Gfarm and NFS with regard to transfer rates.

I. INTRODUCTION

Several types of medical information technology have recently been introduced into hospitals and medical research centers. These systems store and utilize many types and huge amounts of medical information, typically medical image data from MRI, NIRS, CT, etc. The management system used for medical images is the picture archiving and communication system (PACS). PACS stores, browses and manages medical images sent from medical devices such as MRI, NIRS, CT, etc. PACS was designed to use medical images stored at each hospital or facility. However, there has been a recent increase in collaborations between hospitals. When several hospitals collaborate in this way, these medical images must be integrated. One of the easiest ways to integrate distributed medical images or distributed PACS is to integrate all of the medical images onto a central server. However, the amount of medical image information is expected to increase markedly with the future development of medical equipment and systems. In addition to the file size of medical images, the number of

such images is increasing on a daily basis. Based on current trends, it is estimated that over one billion diagnostic imaging procedures will be performed in the USA in 2014, which will generate about 100 petabytes of data[1][2]. Therefore, there are limitations to having a central imaging server for distributed PACS integration. To solve this problem, we focused on distributed PACS that treats and utilizes distributed medical images as one data.

Medical image data standards, such as DICOM (Digital Imaging and communications in Medicine), are very important. DICOM consists of metadata and actual images. The metadata includes various types of medical information, such as inspection items, patient information, etc.

Here, we propose a novel distributed PACS in which only the DICOM metadata are stored on a central server. Actual images are saved on the distributed site. For example, to find a specific image, the user accesses the information stored in the metadata. Thus, it is inefficient to access the entire medical image when user retrieves only the DICOM metadata. Using this method, a high-speed extraction result can be obtained with retrieval of only the metadata.

To implement the proposed system, Gfarm was utilized as a network shared file system[3]. Gfarm stores the metadata of the file as XML data. Therefore, the proposed system can be constructed easily with Gfarm. In the experimental system, the DICOM metadata are stored in Gfarm XML metadata. DICOM is distributed and stored on the storage node. Moreover, DICOM can be retrieved using the XML metadata. A flexible security policy is necessary when DICOM data are shared. DICOM includes information that can specify an individual, such as the patient's name, address, etc. This access to individual information is also controlled by managing the metadata. In the experimental system, individual information is deleted when the medical images are shared.

This paper is organized as follows. Section II presents the back ground of the research. Section III presents the proposed system using DICOM metadata. Section IV describes the

implementation of the experimental system. Section V presents an experiment to verify the system, and Section VI presents the discussion. Section VII provides our conclusions and discusses prospects for future work.

II. BACKGROUND

A. DICOM

DICOM is a typical medical image standard and a network communications protocol for use with medical image diagnosis equipment, such as MRI and CT defined by the ACR (American College of Radiology) and NEMA (National Electrical Manufacturers Association). DICOM is also an international standard (ISO TC-215). DICOM is a container format that can include various types of data, such as image data, and it consists of actual data and information metadata[4]. Fig.1 shows an outline of DICOM.

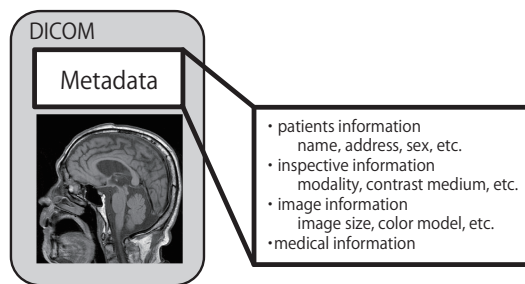


Fig. 1. Outline of DICOM

Various types of information, such as image data size and color model, as well as patient information such as name, address, date of birth and sex are stored in the metadata. The DICOM standard defines more than 3000 terms. When medical images are sorted or found, the information in the metadata is used. However, because information that can specify individual patients is included in the metadata, this raises privacy issues in information sharing.

B. PACS

PACS is now considered the typical standard in the radiology community and is widely implemented even on very large scales[5][6]. PACS have been introduced at a number of hospitals and are used to manage medical images with electronic patient records[7] [8]. However, it has recently become necessary to construct PACS capable of enabling cooperation with outside hospitals. One solution for integrating distributed PACS is to use cloud computing systems, which promise lower costs, high scalability, and high availability[9][10].

C. Distributed file system and Gfarm

1) *Distributed file system:* GFS (Google File System) is a distributed file system developed by Google[11]. On GFS, a large amount of data of PC cluster can be processed. MapReduce is a widely used process to treat such large amounts of data on GFS.

HDFS (Hadoop Distributed File System) is an open source file system which refers GFS[12]. GFS and HDFS are designed to treat data of the order of several million MB, and these systems are not suitable for accessing large amounts of small data. At the same time, methods for these file systems are not versatile because files are operated through the API that is different from standard file systems. On the other hand, Gfarm supports the POSIX standard API and enables decentralized parallel processing. Gfarm shows equivalent read performance and about 30% higher write performance compared to HDFS[13].

2) *Gfarm:* Gfarm is a global distributed file system used to share data and to support distributed data-intensive computing[3]. Gfarm federates local file systems of compute nodes connected on a network, and can be used at various scales, such as LAN, PC clusters, and large-area clusters. Gfarm is provided under an open source license and may be used as Network File System (NFS) and the distributed networked file system Andrew File System (AFS). Gfarm consists of three types of node: metadata servers, which manage the preservation of location information of each file; I/O servers where the main body of data is stored; and clients that access the files. The metadata servers manage file system data of a virtual directory tree and the locations of the actual files, etc., as metadata. First, the client inquires about the position of the I/O server, which stores a file of the metadata server when it access the file which is stored in Gfarm. The client then accesses the I/O server where the file is stored. Fig. 2 shows the Gfarm architecture.

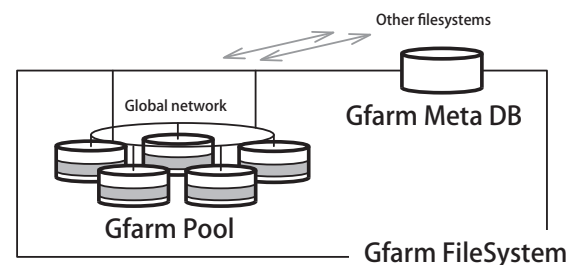


Fig. 2. Gfarm architecture[14]

The Gfarm metadata server not only to manages preservation of the location information of files but also treats metadata as XML data. This mechanism, called XML extended attributes, specifies the related XML file of each saved file, and enables retrieval of the file by XPath. As retrieval becomes possible without directly accessing the file, Gfarm enables high-speed information retrieval in file sharing of the large area.

- Meta data server

The metadata server manages the file system metadata, file open status, I/O server status, global user accounts, group membership information, etc. These information are related to the saved files through XML.

- I/O Server

The I/O server provides file access for a local file system. The client connects directly to the I/O server when accessing file data. The major issue for the I/O server is the maintenance of consistency between the physical file and the file system metadata.

- Client

The client is a host that uses Gfarm. The `gfarm2fs` command is used in the client to mount the Gfarm file system in user space using the FUSE[15] mechanism. After mounting the Gfarm file system by `gfarm2fs` on file system nodes and clients, any existing program can access the Gfarm file system without any modification.

III. PROPOSED DISTRIBUTED PACS

In this chapter, the proposed distributed PACS, which treats metadata and real data of DICOM separately, is described.

A. Outline of proposed system

Here, we assume that several PACS utilized at multiple sights are integrated. As described in Chapter 1, the proposed system involves integration of several PACS with no central PACS. To share large-sized DICOM file over the large-area environment, it is necessary to construct a system to integrate the PACS used in each location. In integrated PACS, it is inefficient to access the whole medical image data when retrieving the DICOM metadata. Thus, it is necessary to separate metadata information from DICOM data.

In the proposed system, DICOM data are stored at each site and DICOM metadata information are stored separately on the metadata server. When metadata are stored, if it is necessary to separate and manage individual information for privacy protection. The access authority to individual information is set according to the authority of the client. It is necessary to construct a system where only clients that have access authority to individual information can retrieve the whole DICOM data.

To achieve this goal, it is necessary to prepare a distributed file system that can be used for metadata, and the metadata information should be managed. The open source network shared file system Gfarm is used for the experimental system discussed here as discussed in the next chapter. Fig. 3 shows the outline of the proposed system.

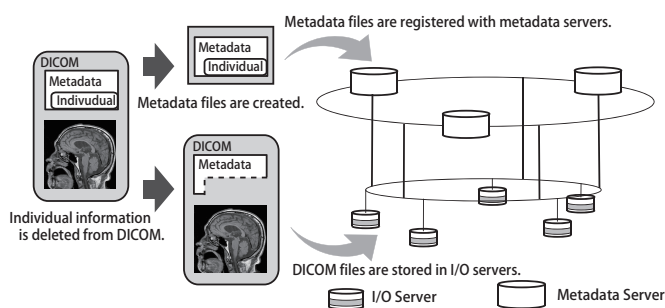


Fig. 3. Outline of proposed system

As shown in Fig. 3, XML are made from the metadata of DICOM and XML metadata is specified for the XML extended attribute of file system. XML is stored on the metadata server. Therefore, the XML metadata alone can be used without the need to access the large DICOM file image information. High-speed retrieval can be achieved because the file system itself retrieves the file. The proposed system stores the DICOM files and allows sharing over a large area. On the other hand, patient information is separated, and the associations with stored DICOM files are preserved by the I/O servers. Account information of the client is set to the DICOM file and access to individual information is controlled appropriately.

The following three DICOM file operations are performed:

- Saving DICOM files

When users store the DICOM file through the client system, the client of the proposed system stores DICOM metadata as XML on the metadata server. DICOM data containing no individual information are stored on the I/O Server. If necessary, individual information is separated from DICOM data.

- Reading DICOM files

When the user accesses the DICOM file, the proposed system reads individual information from the metadata server according to the authority of the client. The proposed system combines the DICOM file obtained from the I/O server with individual information on the client side.

- Retrieval DICOM metadata

The retrieval is performed with the metadata server for the DICOM metadata stored in file system. Moreover, it is possible to use only the DICOM metadata by downloading the XML used for retrieval.

B. Merits of the proposed system

Here, the following four advantages of the proposed system are described.

- Promotion of telemedical care

There is a great deal of interest in telemedicine services with diagnosis based on medical images. The proposed system can facilitate such telemedicine services. The patient may ask a doctor to make a diagnosis using medical images taken at another hospital. In this case, the patient's individual information is not necessary for diagnostic imaging. In the proposed system, DICOM data can be stored without individual information. Thus, the proposed system may contribute to the promotion of telemedicine care because it can be used to acquire only the data necessary for diagnostic imaging.

- Developing Medical treatment Cloud

Patients can receive different examinations using the same medical images obtained at several hospitals if each hospital participates in this system. When medical images are used for the examination, it is currently necessary to obtain permission from the hospital where they were originally taken. The proposed system will allow the

TABLE I
SPECIFICATIONS OF THE SERVER

CPU	Quad-Core AMD Opteron 2.3GHz × 2
Memory	DDR2 667 MHz 8GB
OS	WindowsServer HPC Edition 2007

TABLE II
SPECIFICATIONS OF EACH VIRTUAL NODE

Number of processors	1
Memory	512MB
OS	CentOS 5.5
Kernel	2.6.18-194.el5
gfarm	2.4.0
gfarm2fs	1.2.1

development of a medical treatment cloud system, which will allow patients to easily obtain a second opinion.

- Academic use for medical images
Statistical data of medical images is very important for academic use. Patients' individual information is not necessary for academic use of medical images.
- File system retrieval function
Many items are stored in the DICOM metadata. The DICOM file will contain a number of medical images, and therefore the data size will usually be very large. However, several types of operation can be performed using only the metadata and do not require the actual image data. Therefore, in these cases it is not necessary to download all of the data. The client can quickly retrieve the DICOM metadata without downloading the DICOM data directly using the XML extended attributes, which is the function of Gfarm.

IV. IMPLEMENTATION OF THE PROPOSED SYSTEM

The proposed system was implemented using Gfarm and an experimental system was developed. Gfarm stores XML data as extended XML attributes of the saved file. These mechanisms are suitable for the proposed distributed file system. Gfarm is also distributed under an open source license and is therefore easy to develop. This chapter describes the implemented system.

A. Outline of the implemented system

The actual production system will be located in several remote locations. However, the implemented experimental system was constructed on a virtual server using the Hyper-V function of Windows HPC Server 2008. We constructed an experimental system consisting of one metadata server, two I/O servers, and one client on a virtual node. The next chapter discusses a number of experiments performed using this system. To avoid the influence of communication time and the overhead of each file system is focused, the processing overhead of each file system, the assessment experiment was performed on this virtual system.

Tables I and II show the specifications of the server and of each virtual node, respectively. Fig. 4 shows the flow of the

implemented system.

As shown in the figure, the client divides the metadata and actual images, and these data are stored on Gfarm. When images are searched or derived, the client can obtain the metadata and actual data from Gfarm integrated as DICOM data.

The following sections describe Gfarm and how to use the commands.

B. Introduction to Gfarm

We compiled Gfarm with XML extended attributes enabled. All other settings were left at the defaults, and the common key cryptosystem was used as the authentic method.

C. Commands for the experimental system

We prepared the commands for downloading and uploading of DICOM data. As Gfarm can search a file from XML data, the existing Gfarm search command is used for file retrieval. These commands were prepared on the client system and implemented in Python.

- Uploading DICOM file
We prepared a "dgf_up" command to upload DICOM files to the experimental system. When the client uploads DICOM files to the experimental system using the "dgf_up" command, the experimental system produces XML from the metadata of the specified DICOM files. Individual information is then deleted from the DICOM files. The experimental system stores DICOM image data separately from individual information on the I/O server. The experimental system adds the XML extended attributes using the Gfarm command "gxfattr" for DICOM files stored on the I/O server. The XML extended attributes are managed on the metadata server.
- Downloading DICOM file
We prepared the "dgf_down" command to download DICOM files from the experimental system. When the client acquires DICOM files including individual information using the "dgf_down" command, the client downloads the DICOM files separately from the individual information from the I/O server and XML from the metadata server. The client acquires DICOM files including individual information by reading the individual information from the XML and adding it to the DICOM files on the client side. Moreover, clients that do not have access authority to the individual information can only acquire the DICOM files without the individual information.
- Retrieving DICOM file
The client retrieves DICOM files with XPath using the Gfarm command "gffindxmlattr". Clients that do not have access authority to the individual information can also retrieve DICOM files in the same way using the metadata without the individual information.

V. EXPERIMENTAL ASSESSMENT

As described in this chapter, Gfarm was confirmed to show sufficient performance as a file system for medical images

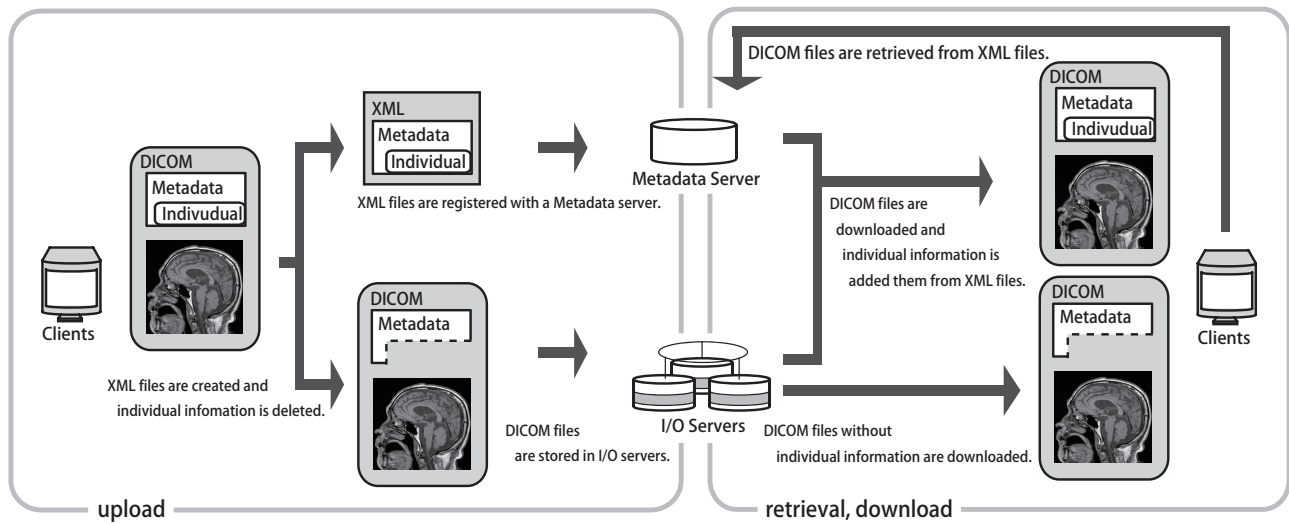


Fig. 4. Flow of implemented system

TABLE III
DICOM AND XML FILE SIZES

File	Size	Number	Sum size
DICOM	135KB	1000	132MB
XML	16KB	1000	16MB

TABLE IV
RESULTS FOR SAVING DICOM AND XML FILES

Process	Gfarm	NFS
real	5m22.507s	5m48.150s
user	0m20.957s	0m05.509s
system	0m56.123s	1m53.721s

and to be useful for separation and retrieval of metadata in the implemented system.

Gfarm was compared with NFS[16], which is a well-known small-scale file sharing system. NFS is not a file system which is designed for large-area environments. However, NFS was compared from the viewpoint of the ease of file access and high-speed performance. In the experimental system, three virtual machines were constructed as an experimental environment for Gfarm. On the other hand, two virtual machines were constructed as an experimental environment for NFS. In NFS, one virtual machine stored the DICOM files minus the individual information and the other stored XML files with the individual information.

Table III shows the sizes of DICOM and XML files used in the experiment. We measured and evaluated the time required to write to the system and delete individual data for thousands of DICOM files and XML files. We measured the time required for processing five times using the “time” command.

A. DICOM and XML file storage

In this section, we compare the processing speeds when DICOM and XML files are stored at the same time on Gfarm and NFS. NFS stored DICOM files on one of two servers, with XML files stored on the other server. Gfarm stored DICOM files on two I/O servers with the XML extended attributes stored on one metadata server. Table IV shows the measurement results for saving 1000 DICOM files and 1000 XML files. In this table, “real” is the total time, “user” is the user CPU time, and “system” is the system CPU time.

TABLE V
RESULTS FOR SAVING XML FILES

Process	Gfarm	NFS
real	2m29.210s	1m23.912s
user	0m14.642s	0m00.098s
system	0m26.630s	0m13.922s

B. XML file storage

In this section, the processing speeds of storing only XML files are compared between Gfarm and NFS. NFS stored XML files on one server. Gfarm added the metadata to the DICOM data already stored on the I/O server. Table V shows the measurement results for saving 1000 XML files.

C. DICOM file storage

In this section, the processing speeds of storing only DICOM files between Gfarm and NFS were compared. Table VI shows the measurement results for saving 1000 DICOM files.

D. XML creation

In this section, the access speed with creating XML from DICOM files was measured. Table VII shows the measurement results for creating XML from 1000 DICOM files.

E. Deleting individual information of dicom

Table VIII shows the access speeds when individual information was deleted from 1000 DICOM files.

TABLE VI
RESULTS FOR SAVING DICOM

Process	Gfarm	NFS
real	2m47.496s	4m03.428s
user	0m00.156s	0m00.149s
system	0m02.200s	1m49.631s

TABLE VII
RESULTS WHEN CREATING XML FROM DICOM FILES

Process	Time
real	8m06.603s
user	7m34.788s
system	0m31.611s

VI. DISCUSSION

As shown in Table IV, the access time for writing is shorter for Gfarm than for NFS. This result suggested that Gfarm has smaller write overhead than NFS. It has also been reported that Gfarm has the same access time as HDFS. Therefore, the proposed system has high file access capability because it uses Gfarm.

As shown in Table V, the Gfarm processing time was longer than that of NFS with regard to writing the XML files on each server. For Gfarm, there is a process for adding XML extended attributes. This process may take some time.

On the other hand, as shown in Table VI, the processing time of Gfarm is shorter than that of NFS. Moreover, it was confirmed that the system CPU time of Gfarm was smaller than that of NFS. These results showed that Gfarm has a mechanism to reduce the overhead.

Tables VII and VIII show the operation times for creating and deleting DICOM data. The proposed system took more than 8 minutes to create 1000 XML files. It also took more than 12 minutes to delete patient information from XML data of 1000 DICOM files. These overhead times are too long and must be reduced in future. The proposed system was implemented using Python. The module involved in data operations will be rewritten in C to improve the speed.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, a distributed PACS was proposed. In the proposed system, the metadata and the actual images of DICOM data are treated differently. For system implementation, the Gfarm network shared file system was used. Through comparative experiments between NFS and Gfarm, we confirmed that the proposed system inherited the characteristics of Gfarm. Thus, the proposed system may have high scalability of access speed and data size. We also confirmed the effectiveness and improved ease of retrieval access control as the proposed system only can access the DICOM metadata.

In future work, we will examine the scalability of large-size files stored in Gfarm. We will also improve the system for use in medical facilities. To achieve this goal, we will examine security and access control mechanisms.

TABLE VIII
RESULTS WHEN DELETING INDIVIDUAL INFO FROM DICOM FILES

Process	Time
real	12m08.176s
user	11m29.293s
system	00m32.916s

REFERENCES

- [1] C.-C. Teng, J. Mitchell, C. Walker, A. Swan, C. Davila, D. Howard, and T. Needham, "A medical image archive solution in the cloud," *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, 2010.
- [2] Frost & Sullivan, "Prepare for disaster & tackle terabytes when evaluating medical image archiving," 2008, <http://www.frost.com>.
- [3] O. Tatebe, K. Hiraga, and N. Soda, "Gfarm grid file system," *New Generation Computing*, vol. 28, pp. 257–275, 2010.
- [4] Herman Oosterwojk, *DICOM Basics Third Edition edition*. OTech Inc, 2005.
- [5] D. Bandon and C. Lovis and A. Geissbuhler and J.-P. Vallee, "Enterprise-wide pacs: beyond radiology, an architecture to manage all medical images," *Academic Radiology 12 (2005)*, pp. 1000–1009.
- [6] Huang HK, "Enterprize pacs and image distribution," *Comput Med Imaging Graphics 2003*, vol. 27, pp. 241–253.
- [7] H. Munch, U. Engelmann, A. Schroeter, and H. Meinzer, "The integration of medical images with the electronic patient record and their web-based distribution," *Acad Radiol 2004*.
- [8] O. Ratib, Y. Ligier, D. Bandon, and D. Valentino, "Update on digital image management and pacs," *Abdom Imaging 2000*, vol. 25, pp. 333–340.
- [9] R. Zheng, H. Jin, Q. Zhang, and P. Chu, "Heterogeneous medical data share and integration on grid," *Proceedings of 2008 International Conference on Biomedical Engineering and Informatics (BMEI '08), IEEE Computer Society Press*, 2008.
- [10] S. Erberich, J. Silverstein, A. Chervenak, R. Schuler, M. Nelson, and C. Kesselman, "Globus medicus — federation of dicom medical imaging devices into healthcare grids," *Stud Health Technol Inform*, vol. 126, pp. 269–278.
- [11] GHEMAWAT S., "The google file system," *Proceedings of 19th ACM Symposium on Operating Systems Principles (SOSP-19)*, 2003, 2003. [Online]. Available: <http://ci.nii.ac.jp/naid/30013225513/>
- [12] The Apache Software Foundation, "Apache hadoop," <http://hadoop.apache.org/>.
- [13] S. Mikami, K. Ohta, and O. Tatebe, "Data intensive distributed computing using mapreduce on gfarm file system," *Information Processing Society of Japan*, vol. 2010, no. 4, 2010.
- [14] G. Datafarm, <http://datafarm.apgrid.org>.
- [15] M. Szeredi, "Fuse : Filesystem in userspace," <http://fuse.sourceforge.net/>.
- [16] B. Callaghan, Pawlowski, and Staubach, "NFS Version 3 Protocol Specification," *RFC 1813*, 1995.

A Framework for Genetic Algorithms in Parallel Environments

Tomoyuki HIROYASU

Department of Life and Medical Sciences
Doshisha University
Kyoto, Japan
tomo@is.doshisha.ac.jp

Ryosuke YAMANAKA

Graduate School of Engineering
Doshisha University
Kyoto, Japan
ryamanaka@mikilab.doshisha.ac.jp

Masato YOSHIMI

Department of Science and Engineering
Doshisha University
Kyoto, Japan
myoshimi@mail.doshisha.ac.jp

Mitsunori MIKI

Department of Science and Engineering
Doshisha University
Kyoto, Japan
mmiki@mail.doshisha.ac.jp

Abstract—In this research, we developed a framework to execute genetic algorithms (GA) in various parallel environments. GA researchers can prepare implementations of GA operators and fitness functions using this framework. We have prepared several types of communication library in various parallel environments. Combining GA implementations and our libraries, GA researchers can benefit from parallel processing without requiring deep knowledge of different parallel architectures. In the proposed framework, the GA model is restricted to a micro-grained model. In this paper, parallel libraries for a Windows cluster environment, multi-core CPU environment, and GPGPU environment are described. A simple GA was implemented with the proposed framework. Computational performance is also discussed through numerical examples.

Index Terms—Genetic Algorithms, Parallel Computing

I. INTRODUCTION

Recently, several types of parallel architecture have come into wide use. For example, calculation with multi-core CPU which more than four cores is not unusual. General purposed GPU becomes also easy to use. In Japan, some of the super-computing centers are open for researchers to use high-end computational resources. We can use the Earth Simulators and will be able to use the next-generation Keisoku supercomputer. However, these parallel architectures have the different configurations. Thus, even when we wish to use the same algorithms, it is necessary to prepare different implementation codes suitable for different parallel architectures. This places a heavy burden on algorithm researchers, because in-depth knowledge of the different parallel architectures is required to run their implementation codes efficiently on parallel machines.

GA is a type of optimization algorithm with multipoint search[1]. GA may find the optimum point even when the landscape of the objective function has multiple peaks. However, GA requires much iteration to find the optimum. This results in high calculation cost. As GA is a multipoint search algorithm, it implicitly has several types of parallelism[2][3][4][5]. Thus, several types of research regarding parallelization of GAs are

existed. Ono et al, introduced the GA model and implementation parallel models of GA should be clarified. As there is parallelism in the GA itself, parallel GA can be performed even on a single process. We call this the logical parallel model. On the other hand, because GA has multiple search points, a single model can be implemented on parallel computers. In this case, an implementation parallel model should be prepared.

In most GA research, these logical and implementation parallel models are not distinguished clearly and are often the same[6][7][8]. When the logical model is closely related to the implementation model, GA users should have deep knowledge of the parallel architectures on which their parallel GAs are running. At the same time, as the logical model and implementation model are closely related, different parallel codes are required for different parallel machines. Therefore, it would be of great benefit if GA users were not required to have such deep knowledge of novel parallel architectures to run their GAs in parallel.

Here, we propose a parallel environment framework for GA that adopts the micro-grained model as an implementation model. GA researchers prepare the implementations of GA operators and fitness functions using the proposed framework. We are preparing parallel communication libraries for this framework. Using GA implementations and these libraries, GA users can derive efficient parallel GA codes without requiring specific knowledge regarding to parallel architectures. Thus, the proposed framework improves the productivity of GA users.

We constructed a system for Windows cluster with Windows Communication Foundation (WCF) in C# with multi-threading in C language for three types of parallel environment, i.e., cluster, multi-core CPU, and GPU. In addition, we verified the system through the preliminary experiments.

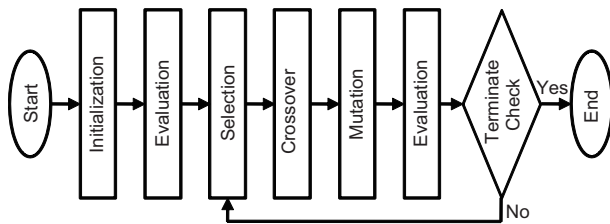


Fig. 1. Flowchart of GA.

II. GENETIC ALGORITHM

A. Overview

The GA is an optimization algorithm that mimics natural evolution with variation and adaptation to the environment. In evolution processes in nature, an individual that is better adapted to the environment among a group of individuals forming a certain generation survives at a higher rate, and leaves offspring to the next generation. In the GA concept, the computer finds an individual that is better adapted to the environment, or a solution that yields an optimum value to an evaluation function, by modeling the mechanism of biological evolution. Figure 1 shows a typical flowchart of GA.

The GA applies genetic operations, crossover and mutation, to each individual in the population to produce new individuals. These individuals are evaluated and the GA selects superior individuals for the next generation. The GA searches for a solution by repeating this series of operations until the termination condition is met. The evaluation time is the same as the number of populations. GAs have good performance in parallel environments because they have data-level parallelism.

B. Parallel Model of GA

The GA is able to parallelized because it searches multiple points and repeats sampling. Parallel models of GA can be divided into coarse-grained and micro-grained models

1) *Coarse-grained model*: The coarse-grained model is generally called a distributed population model. This model splits the population into multiple subpopulations, which are then searched. Therefore, several individuals in several subpopulations are moved into other subpopulations. This operation is called the migration. Figure 2 shows the flow of the coarse-grained model. This model uses computational resources effectively, because it connects to computational nodes only during migration. In addition, this model changes performance of the search compared to a serial algorithm.

2) *Micro-grained model*: Evaluations account for a large share of total execution time in complex of objective problems. The micro-grained model is based on the general concept of parallelization. This model is a master-slave model. A master processor executes other genetic operations besides evaluation.

Evaluations are executed by slave processors. A master processor sends individuals that should be evaluated. Slave processors evaluate these individual, and return them to the

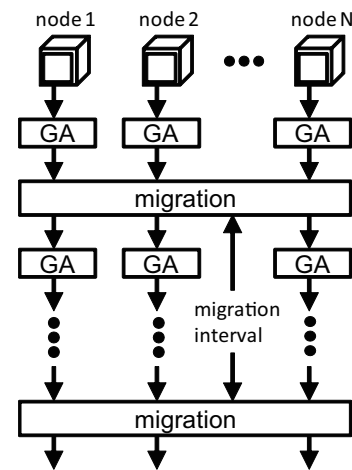


Fig. 2. Coarse-grained model.

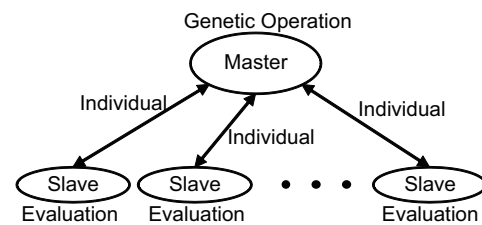


Fig. 3. Micro-grained model.

master processor. Figure 3 shows the flow of micro-grained model. This model shows inferior parallelization performance compared to the coarse-grained model, because it must have many connections and the master processor uses a CPU. In addition, this model does not alter the search performance compared to a serial algorithm.

III. PROPOSED FRAMEWORK

A. Background

In conventional parallel GA research, the proposed GAs are often fully connected to the particular parallel environment. In these cases, the proposed GAs cannot be used in the different parallel environments. Advanced programming techniques are required to use computational resources with GPGPU and many cores. As new architectures appear, more GA users will be required to make implementations for them. This represents a burden on GA users. The coarse-grained model has important differences from the micro-grained model. The coarse-grained model has few connections and can use parallel environments. However, when this model is adopted, the designed GA and the search performance are changed. The micro-grained model, on the other hand, does not alter the search performance are changed. At present, users adopt the coarse-grained model to make effective use of parallel environments. Therefore, GA development is limited.

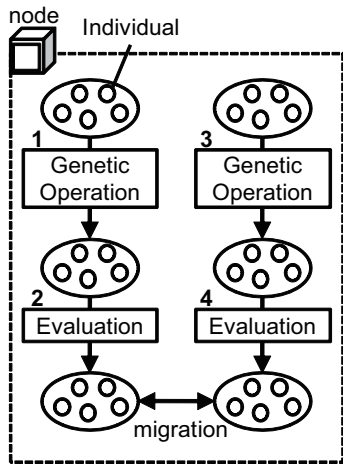


Fig. 4. Island model with serial implementation.

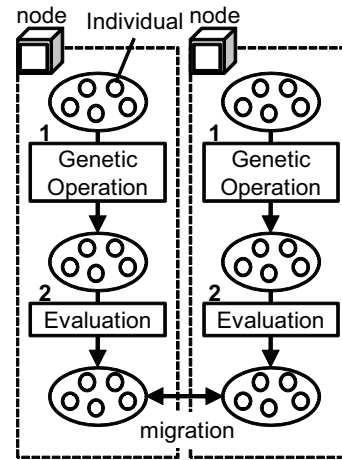


Fig. 5. Island model with parallel implementation.

B. Requirements

To overcome the disadvantages discussed in the previous section, a framework is proposed with the following requirements:

- Systematization of parallel models
- Adoption of micro-grained model
- Standardized interface

These requirements are described in more detail below.

1) *Systematization of Parallel Models*: There are two aims of parallelization in GA. The first is to parallelize algorithms for improved search performance. The second is to parallelize the implementation to reduce computational time. We must declare two parallelization that parallel implementation does not confine algorithms of GA. We define them as a logical model and an implementation model. Figure 4 shows a GA that adopts the island model as a logical model and serial model as an implementation model. Figure 5 shows a GA that adopts the island model as both the logical model and as the implementation model. The logical model searches in parallel; however, it can also be implemented in serial. In addition, the logical model is confined by the limits of implementation model.

2) *Adoption of Micro-grained Model*: It is necessary for GA users to use an arbitrary algorithm with which any and all GAs can benefit from parallel processing. The micro-grained model is adopted as the implementation model. Therefore, the GA can be implemented without changing the logical model.

3) *Standardized Interface*: To reduce the burden on GA users, it is necessary that they should be able to use any and all parallel environments with a common interface.

C. Overview of Proposed Framework

The purpose of the proposed framework is to allow GA users to perform parallel processing with the micro-grained model as an implementation model of GA, without programming techniques for parallel processing. Figure 6 shows

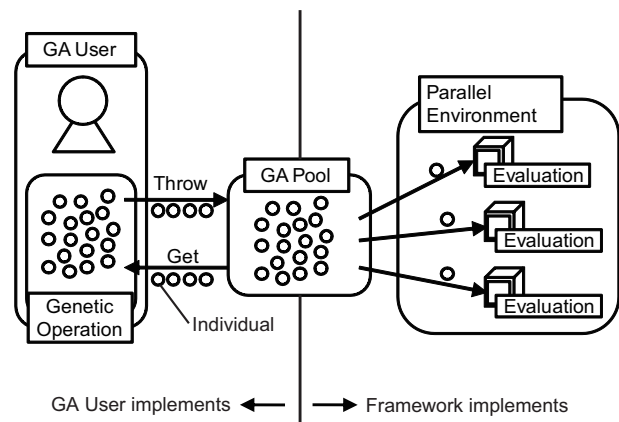


Fig. 6. Concept of the framework.

an overview of the framework. The framework introduces the concept of the GA Pool as the interface. GA users throw individuals into the GA Pool. Thus, they are able to get evaluated individuals from the GA Pool. The GA Pool evaluates thrown individuals with parallel environments. The framework supplies an implementation for use of parallel environments. GA users can construct an arbitrary logical model and implement GA operations besides evaluation part. They implement the evaluation part with the prepared template. This template has arguments and return value of the function of the evaluation. This template hides implementation of a particular connection and scheduling for the task of evaluation from the GA user. Thus, GA users can construct GAs adapted to parallel environments without requiring knowledge regarding connections and scheduling jobs.

D. GA Pool

The GA Pool is composed of two queues as shown in Figure 7. These queues are the throw queue, which puts thrown individuals, and the get queue, which puts evaluated individuals. When individuals are put, a thread that monitors the throw queue sends individuals to computational resources

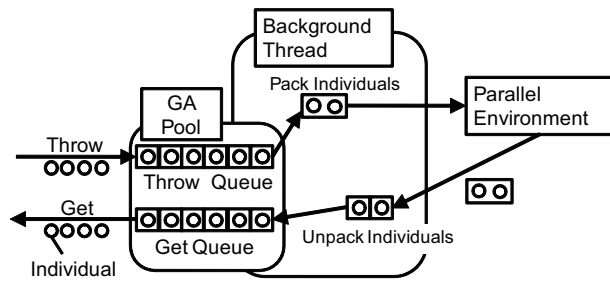


Fig. 7. Constitution and function of GA Pool.

TABLE I
ARCHITECTURE OF A NODE ON THE WINDOWS CLUSTER.

OS	Windows HPC Server 2008
Memory	8 GB
CPU	AMD Opteron 2356(Quad) × 2

and calculates the evaluation. The thread executes connections adapted to the architecture of the computational resources in the parallel environment. When the thread puts evaluated individuals into the get queue, it unpacks them one by one.

IV. EVALUATION

A. Confirmation of Parallelism with Framework

This section confirms the parallelism of GAs constructed with the proposed framework in a cluster environment.

1) *Computational Environments*: A Windows cluster running Windows HPC Server was used as a distributed memory environment. The communication infrastructure was Windows Communication Foundation (WCF)[9] with C#, and we used a novel parallel model different from MPI. This parallel model is based on Service Oriented Architecture (SOA)[10][11][12][13]. Only function implementation is located on the computational nodes as slaves. The client machine acting as a master calls the function to control jobs interactively. WCF does not share sources between master and slave processors. Therefore, it has the good expandability. In addition, WCF adapted the micro-grained model such that the slaves execute evaluation only, because slave processors have only the function evaluation. The Windows cluster is able to view a core as a computational resource. This section discusses confirmation of the parallelism of GAs constructed with the proposed framework with 16 cores on 2 machines, as shown in Table I. Table II shows the parameters of the GA used in this section.

2) *Results*: Figure 8 shows the relation between the number of computational nodes and execution time. List 1 shows pseudo-code of a simple GA constructed with the proposed framework. Lines 4, 8, 11, and 20 in List 1 are descriptions for using the framework. GA users add only four descriptions and can reduce the execution time by increasing the calculation resources as shown in Figure 8.

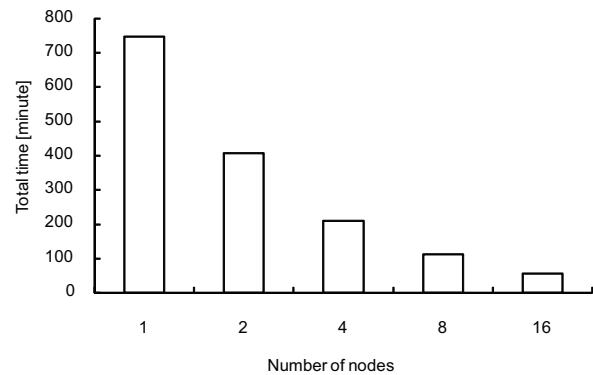


Fig. 8. Relation between number of computational nodes and execution time.

List 1. Simple GA constructed with the proposed framework

```

1 // initialization of population
2 InitPopulation();
3 // initialization of framework
4 Initialize(POPULATION_SIZE, MAX_GENERATION);
5 for (int i = 1; i <= MAX_GENERATION; i++) {
6     for (int j = 0; j < POPULATION_SIZE; j++)
7         // throw individuals to GA Pool
8         Throw(individual);
9     for (int j = 0; j < POPULATION_SIZE; j++)
10        // get individuals from GA Pool
11        individual = Get();
12    // selection
13    population = selection(population);
14    // crossover
15    crossover(population);
16    // mutation
17    mutation(population);
18 }
19 // Finalization of framework
20 Finalize();

```

B. Verification of connection performance with along to data size

This section evaluates the connection performance of a cluster, multi-core CPU, and GPU. We verify the influences of data volume and number of connections on execution time with changing numbers of individuals in a connection. There are several parameters in algorithms and parallel libraries, and these parameters should be tuned optimally to achieve high parallel efficiency. In a distributed memory environment, communication overheads are large. In a shared memory environment, it is necessary to consider the limits of memory based on the processor architecture. Here, we use a system that controls data volume and number of connections when the master processor communicates with the slave processors. In particular, the system controls the number of individuals in

TABLE II
PARAMETERS OF GA.

Parameter	Value
Population Size	64
Gene Length	41
Max Generation	32
Optimization Problem	HRE[14]
Logical Model	Simple GA[1]

TABLE III
ARCHITECTURE OF NODE WITH MULTI-CORE CPU.

OS	Debian 5.0.8
Memory	16 GB
CPU	AMD Opteron 2423 2.0 GHz (Six-Core) × 2

TABLE IV
ARCHITECTURE OF NODE WITH GPU.

OS	CentOS
Memory	16 GB
CPU	AMD Opteron 2356 2.3 GHz (Quad) × 2
GPU Memory	512 MB
GPU	NVIDIA GeForce GTX250 1.84 GHz (128-Core)

a connection as shown in Figure 7. During the simulation, the best data volume and number of connections can be determined dynamically.

1) *Environments*: The cluster environment is the same as the system used in section IV-A. A multi-core CPU and GPU are used as shared memory environments. We used C language on multi-core CPU and CUDA with NVIDIA on GPU. In shared memory environments, multiple threads in relation to the number of cores are generated for parallel processing. GA users need not send a program of the evaluation module to computational resources. We use computational resources to compare each environment. One node is used for a Windows cluster and one thread is used for multi-core CPU and GPU.

Tables III and IV show the specifications of multi-core CPU and GPU.

Table V shows the parameters of the GA used in this evaluation. A total 512 data (4000 bytes each) are sent to the computational resources, because the population size is 512 and the data volume of an individual is 4000 bytes. Power-of-two data are sent, and we record the execution time.

2) *Results*: Figures 9, 10, and 11 describe the relation between data volume in a connection and execution time in each parallel environment. The values in these graphs are the medians of 100 independent trials. As show in each figure, the data volume was confirmed to affect the execution time. In addition, we confirmed that the optimal data volume that reduces the running time to the greatest extent is different for each architecture. The optimal values are 64 × 4000 bytes for the cluster, 8 × 4000 bytes for the multi-core CPU, and 16 × 4000 bytes for the GPU is.

V. DISCUSSION

As shown in Figures 9 - 11, it was confirmed that the minimum execution time according to data volume was different between the cluster, multi-core CPU, and GPU environments.

TABLE V
PARAMETERS OF GA.

Population Size	512
Gene Length	1000
Data Type of a Gene	int (4 bytes)

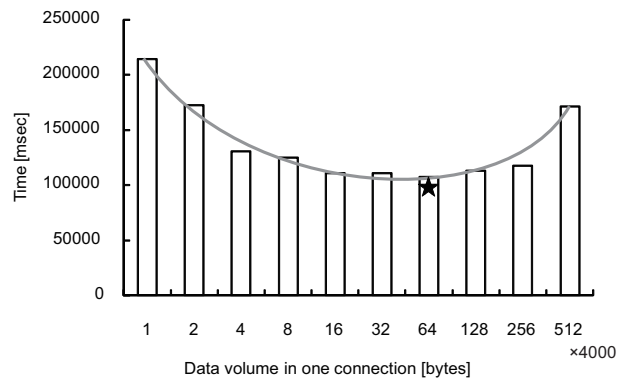


Fig. 9. Relation between data volume in one connection and execute time on Windows cluster.

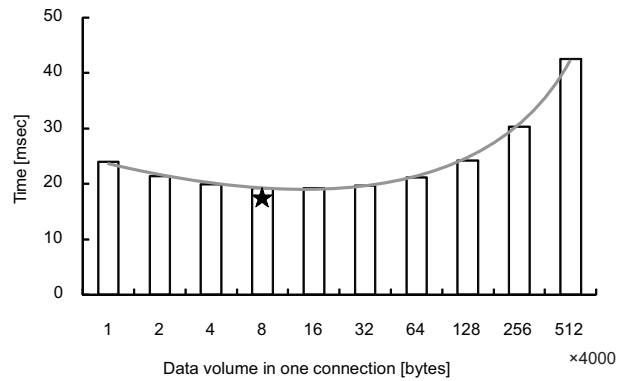


Fig. 10. Relation between data volume in one connection and execute time on multi-core CPU.

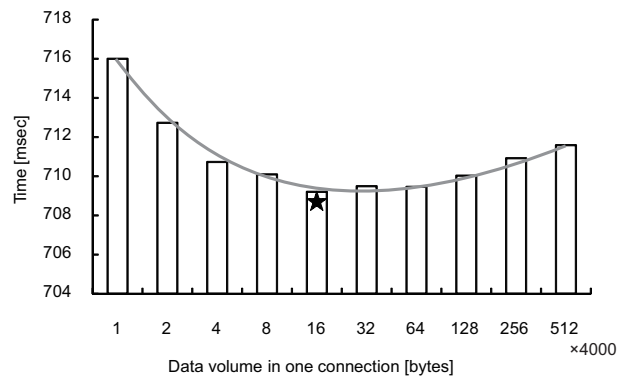


Fig. 11. Relation between data volume in one connection and execute time on GPU.

Table VI shows the differences between the max and min values of execution time in each environment. As shown in

TABLE VI
DIFFERENCES BETWEEN MAX AND MIN TIMES.

Environment	Difference [msec]
Cluster	107170
Multi-core CPU	23.1068
GPU	3.52358

this table, the difference is smallest in GPU and the largest in cluster. This suggests that tuning has a greater effect in distributed memory environments. However, the tuning affects are also different even for shared memory environments. This evaluation used only homogeneous parallel environments. However, these results show that changing data volume has a greater influence in heterogeneous parallel environments.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a framework for GAs in parallel environments. GA researchers can prepare implementations of GA operators and fitness functions using this framework. We have prepared several types of communication library for use in various parallel environments. Combining the GA implementations and our libraries, GA researchers can benefit from parallel processing without requiring deep knowledge regarding parallel architectures. In the proposed framework, the GA model is restricted to a micro-grained model. In this paper, parallel libraries for Windows cluster environment, multi-core CPU environment, and GPGPU environment were prepared.

For the Windows cluster, parallel communication libraries were prepared with WCF and C#. Using the libraries and the framework, GA researchers can implement the parallel processing part with only four descriptions. In addition, we verified data volumes and number of connections on the Windows cluster, multi-core CPU, and GPU with a system that changes the number of individuals in a connection. The results indicated that the best number of individuals in a connection differs according to the architecture.

In future work, a mechanism to find the best number of individuals and to tune it dynamically will be implemented in the libraries. In addition, we will also attempt to prepare other parallel libraries for other parallel architectures.

REFERENCES

- [1] David E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
- [2] T. Starkweather, D. Whitley, and K. Mathimas, *Optimization using Distributed Genetic Algorithms*, Parallel Problem Solving form Nature, 1991.
- [3] H. Mühlenbein, "Parallel genetic algorithms, population genetics and combinatorial optimization," in *Parallelism, Learning, Evolution*, vol. 565 of *Lecture Notes in Computer Science*, pp. 398–406. Springer Berlin / Heidelberg, 1991.
- [4] C. Belding Theodore, "The Distributed Genetic Algorithm Revisited," *Proc. 6th International Conf. Genetic Algorithms*, pp. 114–121, 1995.
- [5] 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.
- [6] Lim D., Ong Y. Soon, Jin Y., Sendhoff, B., and Lee B. Sung, "Efficient hierarchical parallel genetic algorithms using grid computing," *Future Generation Computer Systems*, vol. 23, no. 4, pp. 658–670, 2007.
- [7] J. Ming Li, X. Jing Wang, R. Sheng He, and Z. Xian 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*, 2007, pp. 855–862.
- [8] 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.
- [9] "Windows Communication Foundation," <http://msdn.microsoft.com/en-us/library/dd456779.aspx>.

- [10] "Understanding Service-Oriented Architecture," <http://msdn.microsoft.com/en-us/library/aa480021.aspx>.
- [11] M.P. Papazoglou and D Georgakopoulos, "Service-Oriented Computing," *Communications of the ACM*, vol. 46, no. 10, pp. 25–28, Oct. 2003.
- [12] W. Zhang and G. Cheng, "A service-oriented distributed framework-wcf," *Web Information Systems and Mining, International Conference on*, vol. 0, pp. 302–305, Nov. 2009.
- [13] Alaa M. Riad, Ahmed E. Hassen, and Qusay F. Hassen, "Design of SOA-based Grid Computing with Enterprise Service Bus," *International Journal on Advances in Information Sciences and Service Sciences*, vol. 2, no. 1, pp. 71–82, Mar 2010.
- [14] Y. Kosugi, A. Oyama, K. Fuji, and M. Kanazaki, "Conceptual Design Optimization of Hybrid Rocket Engine," *Proceedings of Space Transportation Symposium*, 2009.

An Intelligent Lighting System to Realize Individual Lighting Environments Based on Estimated Daylight Distribution

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

Tomoyuki HIROYASU

Department of Life and Medical Sciences
Doshisha University
Kyoto, Japan
tomo@is.doshisha.ac.jp

Masato YOSHIMI

Department of Science and Engineering
Doshisha University
Kyoto, Japan
myoshimi@mail.doshisha.ac.jp

Hiroyuki YONEMOTO

Graduate School of Engineering
Doshisha University
Kyoto, Japan
hyonemoto@mikilab.doshisha.ac.jp

Abstract—When we introduced a lighting system to realize individual lighting environments into real office environments, difficulties arose in placing illuminance sensors on users' workplanes. This study hence proposes a new approach to control a lighting system intended to realize individual lighting environments without placing illuminance sensors on users' workplanes. This system uses illuminance sensors for measuring not the illuminance on workplanes but that of daylight: it optimizes lighting based on simulations for different luminous intensities of lighting and patterns of daylight illuminance distribution which are estimated from measurements by daylight illuminance sensors. An experiment to converge illuminance at target positions into target illuminance levels was conducted in a setting with 15 fluorescent lights and 9 illuminance sensors, which was intended to simulate a real office environment. The result indicated that such a system can realize illuminance levels required by individual users with minimum power consumption responding to changing daylight conditions.

Index Terms—lighting, intelligent lighting system, illuminance distribution, daylight, optimization, illuminance sensor

I. INTRODUCTION

With the development of electronic parts and information technologies, microcomputer chips are now built into many machines. In this context, there have been many attempts to develop an intelligent system which enables the machine itself to autonomously control its operation to suit user or environmental requirements. In the field of lighting and air conditioning, however, the introduction of intelligent systems has been rather slow compared to other products from such concerns as installation costs. Yet at last in recent years, attempts of intelligent designs have increased also in lighting systems, intended, for instance, to realize lighting patterns meeting different user requirements or to minimize energy consumption. One example is a residential lighting fixture with automatic brightness adjustment function by sensor[1]. In this

system, an illuminance sensor built into the lighting fixture detects reflection by the surrounding surfaces and natural daylight so that, based on the measurement, the system may control the luminous intensity of the lighting to keep the illuminance within the area at a certain level. Such a system can prevent the luminous intensity from being higher than necessary so as to minimize energy consumption.

This lighting system, however, cannot provide brightness (illuminance) at the level and the point as desired by the user, as long as the illuminance sensor is positioned on the lighting fixture. On the other hand, it has been reported that providing the illuminance most appropriate to the task of each worker is an effective choice for the improvement of office environment[2]: to realize energy efficiency and to improve optical environment for each worker, the use of task-ambient lighting systems will be an effective approach.

But in reality, task-ambient lighting systems are not widely accepted in Japanese offices because (1)typical office buildings are equipped with ceiling lighting fixtures which can ensure a desktop illuminance of 750 lx without a task-ambient lighting, and (2)most companies are not willing to pay additional costs for purchasing task-ambient lightings, as well as consider that task-ambient lighting systems spoil the visual impression of the office.

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[3], [4]. With this intelligent lighting system, each user specifies a target illuminance level for an illuminance sensor which is to be placed on the workplane, then the system will realize the target illuminance level. The intelligent lighting system, composed of a lighting fixture, a control device, an illuminance sensor and a wattmeter, can re-

alize any lighting patterns as required by the user independent of electrical wiring. The intelligent lighting system has proven successful in our laboratory experiments[5].

Toward the commercialization of our intelligent lighting systems, currently verification experiments are underway in several offices in Tokyo[6]. The results so far indicate that there are cases where it is difficult to place an illuminance sensor on the user's workplane. In this study, we will propose a new approach for controlling intelligent lighting systems to realize an individualized lighting environment without a need to place an illuminance sensor on the user's workplane. Further, an operational experiment under an environment simulating a real office is conducted to verify the effectiveness of the proposed system.

II. INTELLIGENT LIGHTING SYSTEM

A. Construction of Intelligent Lighting System

The intelligent lighting system, as indicated in Fig.1, is composed of lights equipped with microprocessors, portable illuminance sensors, and electrical power meters, with each element connected via a network.

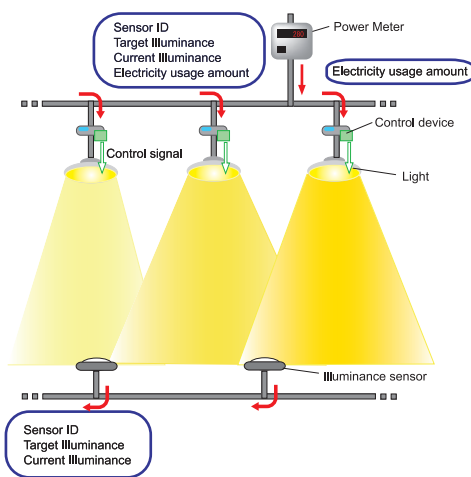


Fig. 1. Configuration of Intelligent Lighting System

Individual users set the illuminance constraint on the illuminance sensors. At this time, each light repeats autonomous changes in luminance to converge to an optimum lighting pattern. Also, with the intelligent lighting system, positional information for the lights and illuminance sensors is unnecessary. This is because the lights learn the factor of influence to the illuminance sensors, based on illuminance data sent from illuminance sensors. In this fashion, each user's target illuminance can be provided rapidly.

The most significant feature of the intelligent lighting system is that no component exists for integrated control of the whole system; each light is controlled autonomously. For this reason, the system has a high degree of fault tolerance, making it highly reliable even for large-scale offices.

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)[7], 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) A regression analysis is conducted and the level of influence is estimated.
- 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) Steps 2~9 are one search operation of the luminance value, which is repeated.

A search operation process (requiring about 2 seconds) consists of steps 2) through 9) 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 target points
 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. Fig.2 shows how the intelligent lighting system is used in an office of Mori Building Co., Ltd. (Roppongi Hills Mori Tower). As shown in the photo, an illuminance sensor is placed on the workplane and the user presets the target illuminance. This enables the intelligent lighting system to realize the targeted brightness on the user's workplane using the control algorithm described in the preceding chapter.

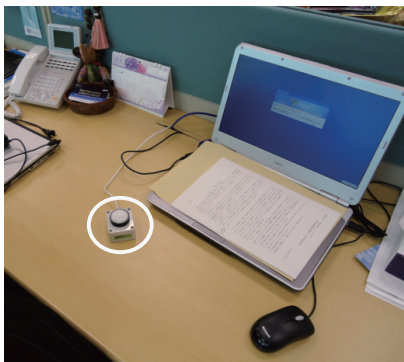


Fig. 2. An experimental intelligent lighting system in a real office

In our experiments in some offices, however, it was found difficult to place illuminance sensors on user workplanes due to mounting documents. In those cases, illuminance sensors were placed at such points as a corner or a partition top. Fig.3 shows an example of an illuminance sensor placed on top of a partition. Positioning the illuminance sensor this way makes it impossible for the system to realize the target illuminance on the user's workplane.



Fig. 3. A situation prohibiting the placement of an illuminance sensor on the workplane (example)

Therefore, to solve this problem while maximizing the workspace available for the user, we propose a new control algorithm for intelligent lighting systems.

IV. OPTIMAL CONTROL OF LIGHTING BASED ON ESTIMATED DAYLIGHT DISTRIBUTION PATTERNS

A. Configuration of the proposed system

As mentioned in the preceding chapter, it was found difficult in some real office environments to place illuminance sensors on users' workplanes. Hence, we propose a new system to realize the desired illuminance at any given point specified by each user with minimum power consumption without placing an illuminance sensor on the user's workplane. To realize the target illuminance at the workplane which does not have an illuminance sensor, the system estimates illuminance there by simulation. Further, to maximize the accuracy of simulation, illuminance sensors are set in readily available spaces such as partition tops, and the system estimates patterns of daylight illuminance based on the distribution of illuminance measurements. Unlike earlier intelligent lighting systems, this system requires data on user and illuminance sensor positions for making simulations. Here, the point where the target illuminance should be realized is typically the user's workplane. Therefore the positions of target points where certain illuminance levels are desired by users will be readily known since desk positions are usually fixed.

The proposed system will be composed of lighting fixtures, illuminance sensors and a central control device. In the proposed system, an optimum lighting pattern is found based on

simulations using illuminance sensor measurement data, then the luminous intensity of each lighting is determined. Since this makes it impossible to use a distributed control approach with control devices built into each lighting fixture, a central control device is used unlike our earlier intelligent lighting systems. Illustrative configuration of the proposed system is shown in Fig.4, which is a view of a room from the top.

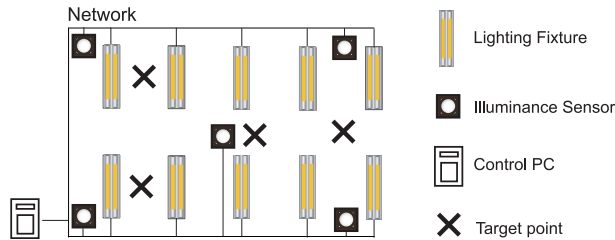


Fig. 4. Illustrative configuration of the proposed system

B. Proposed system control

The proposed system estimates the effect of light from sources other than the lightings under control, such as daylight, based on the measurement data obtained from all illuminance sensors in different positions within the room. Then, in view of the distribution of illuminance from daylight, it determines an optimum lighting pattern based on a simulation. For this simulation, the Stochastic Hill Climbing method (SHC) is used as it was in preceding intelligent lighting systems. Although SHC tends to find a local optimum solution if the objective function is multimodal, studies have indicated that a good solution can be obtained with SHC by limiting the area of neighborhood to an appropriate range, which is defined as a range of generation of next solutions.

By using this method, the system will be able to provide the desired illuminance results without setting illuminance sensors on users' workplanes. The flow of control of the proposed method is shown below. Here, illuminance sensor positions and target positions are given initially.

- 1) Calculate the optimum luminous intensity of each lighting to realize the target illuminance at target positions with minimum power consumption based on illuminance calculation, then turn on each lighting with the calculated luminous intensity (the luminous intensity from daylight is assumed to be 0 lx).
- 2) Obtain illuminance data from illuminance sensors.
- 3) Calculate the difference between the calculated illuminance and the actual illuminance measured at each illuminance sensor position, which should be the illuminance from daylight at the sensor position.
- 4) Estimate the distribution of daylight in the entire room based on the daylight illuminance values obtained from the above calculation, and then estimate the illuminance from daylight at each target position.
- 5) For each target position, optimize the luminous intensity of the lighting so as to bring its illuminance as close as

possible to the difference between the target illuminance and the illuminance from daylight.

- 6) Repeat steps 2) through 5).

Using this method, even when illuminance sensor positions do not coincide with the points where the desired illuminance should be realized, the system can realize the target illuminance values responding to changing daylight conditions with minimum power consumption. Just as in our previous intelligent lighting systems, the objective function of Eq.(1) is used, where illuminance L_c is the sum of the illuminance from the lighting (lighting illuminance) and the illuminance from natural daylight (daylight illuminance).

$$L_c = L_l + L_d \quad (3)$$

L_l : Illuminance from lighting

L_d : Illuminance from daylight

Lighting illuminance is calculated using an illuminance simulator described in the following section, while daylight illuminance at any given point is calculated using a daylight simulator described in the following section. By using these two simulators, the optimization problem expressed by Eq.(1) is solved to realize lighting control.

C. Illuminance simulator

To calculate illuminance at a given point in a given room, different approaches have been studied including the point-by-point method and the lumen method calculations using Monte Carlo method[8], which are known to be capable of realizing a high level of accuracy. Still, to ensure a high level of accuracy with these methods require defining many parameters such as the luminous flux of lighting, maintenance factor, luminous intensity distribution curve or reflection by walls, of which values are not readily known in most real-world environments.

Therefore, to enable highly accurate simulations in a simplified method, the proposed approach uses only a limited number of parameters. Given that the target points where a certain illuminance level is required are on fixed workplanes, here we need to simulate only illuminance at specific positions rather than at any arbitrary positions. In the proposed approach, the actual illuminance at a particular point is measured with the relevant lighting illuminated at a particular luminous intensity, and the level of influence by the lighting at that point is stored on a database. Then based on this value, illuminance at that particular point under any given lighting pattern can be calculated. At the illuminance sensor positions, the daylight illuminance can be calculated as a difference between the measured illuminance and the lighting illuminance.

D. Daylight simulator

Different approaches have been studied to calculate daylight illuminance distribution[9], [10], which use such factors as the position of the sun, the amount of clouds, and the transmittance of the windowpane material to calculate daylight distribution patterns. Yet they either assume an environment without a window blind or require detailed data on the effects of blinds.

Also, dust on blinds may change the reflectance of the blind to spoil the accuracy of simulation.

Therefore in the proposed method, illuminance sensors are positioned in spaces where they can be placed readily and the daylight illuminance distribution function is estimated using the least square method based on the daylight distribution assessed as described above.

E. Derivation of a model equation

Plenty of illuminance sensors were set in the experimental environment as shown in Fig.5 and daylight illuminance was measured by each sensor. A model equation was derived based on the daylight illuminance measurements on sunny, cloudy and rainy days over a period from October to January.

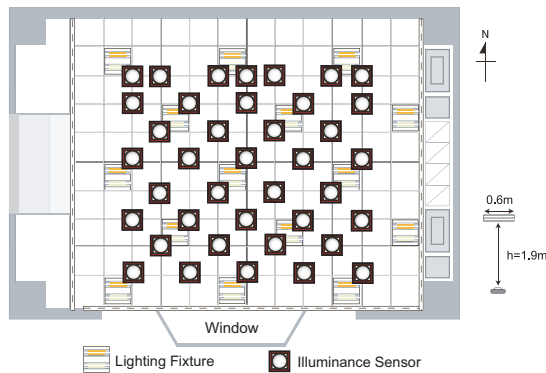


Fig. 5. Daylight illuminance measurement system

After making many trials and errors, we derived a model equation which best expresses the indoor daylight illuminance distribution. Eq. (4) is the derived model equation. In the proposed method, the daylight illuminance distribution function is estimated based on the model equation, when positional coordinates are (x,y) and the daylight illuminance at that position is z .

$$z = \beta_0 + \beta_1 x^4 y^3 + \beta_2 x^3 y^4 + \beta_3 x^3 y^2 + \beta_4 x^2 y^4 + \beta_5 x y^3 + \beta_6 x y^2 + \beta_7 y \quad (4)$$

V. SUMMARY OF AN OPERATIONAL EXPERIMENT

An operational experiment was conducted for a total of 9 hours between 7:00 and 16:00 on December 19, 2010, which was a sunny day. The proposed system was constructed and its validity was tested for verification. Illuminance sensors were set at regular intervals and the points for which users will specify desired illuminance levels (hereinafter called “target points”) were defined.

The experiment used 9 illuminance sensors and assumed 5 users with target points arranged as shown in Fig. 6. The target points included points A, B, C, D and E, for which the target illuminance was set at 400 lx, 500 lx, 550 lx, 600 lx and 700 lx respectively.

Illuminance data were taken every second and lightings were turned out once in every minute. Since the purpose of turning out lights here is only to allow comparison between the

daylight illuminance distribution as estimated by the proposed system and the actual distribution of illuminance from daylight, they were never turned out while operating the system under the proposed method. For the experiment, window blinds were arranged 45 degrees outward and neutral white fluorescent lamps were used of which luminous intensity was variable between 401 cd and 1336 cd.

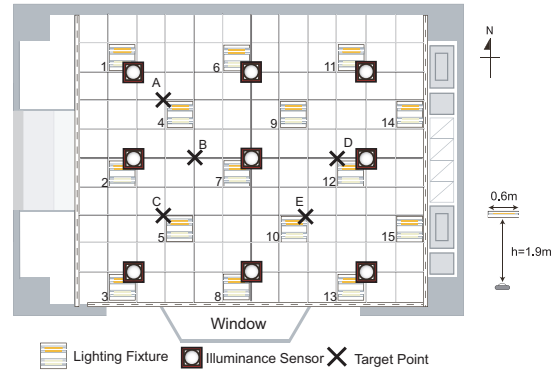


Fig. 6. Experimental environment

In the operational experiment, the lighting pattern was changed every 30 seconds to verify whether the target illuminance levels are constantly realized at target points.

VI. EXPERIMENT RESULTS AND DISCUSSIONS

Fig.7 shows history of the illuminance data at target points A, B, C, D and E. Fig.8 shows history of the daylight illuminance data, measured once in every minute with lights turned out. Fig.9 shows history of the luminous intensity data of lights 4, 5, 7, 10 and 12 which are located near some target point.

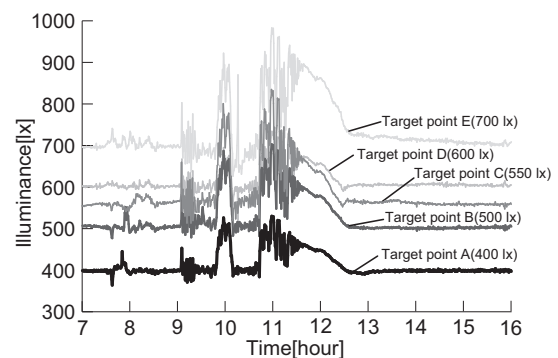


Fig. 7. History of the measured illuminance data

Fig.7 indicates that target luminance levels were constantly achieved after 12:30. There were, however, some periods when target illuminance levels were not achieved: particularly around 10:00 and from 11:00 until around 12:20, the actual illuminance levels were higher than target values. Noting that, let us examine Fig.8 and Fig.9: Fig.8 indicates that in those periods when target illuminance was not achieved, the effect of daylight illuminance was significant. Also, as one can see from Fig.9, the luminous intensity levels of the lightings were kept

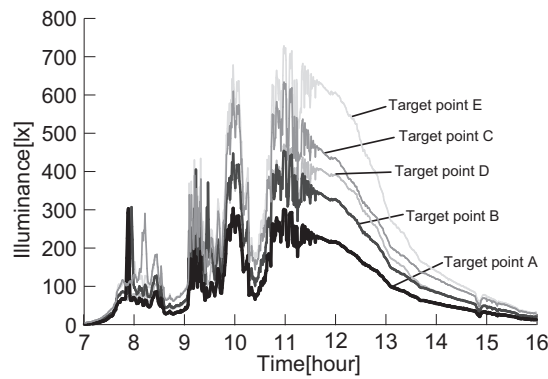


Fig. 8. History of the measured daylight illuminance data

at a minimum level during such periods. These results indicate that when actual daylight illuminance is too large, it was simply physically impossible to realize the target illuminance levels. For this, from the history of the measured illuminance data in the periods between 7:00 and 9:40 and after 12:30, we can learn that the luminous intensity of each lighting changes as the measured daylight illuminance changes, and thus the target illuminance values can be achieved.

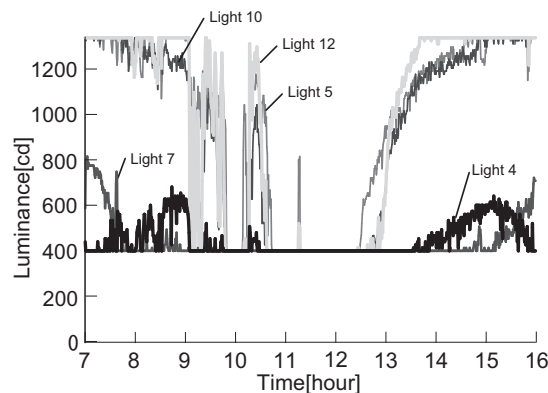


Fig. 9. History of the luminous intensity in the proximity of target points

Next, we will examine the energy efficiency of the proposed method. Fig.10 shows history of the luminous intensity data of lightings number 9 and 11 which are distant from target points.

Fig.10 indicates that lightings distant from target points illuminated at a minimum intensity, demonstrating the energy efficiency of the proposed system. These results demonstrated that the proposed method can provide desired illuminance levels on workplanes without setting illuminance sensors on those workplanes while saving energy.

VII. CONCLUSION

In this paper, we proposed a new control algorithm for an intelligent lighting system which realizes desired illuminance levels on workplanes based on illuminance distribution patterns estimated from illuminance data obtained by illuminance sensors placed on readily available spaces such as partition tops instead of workplanes.

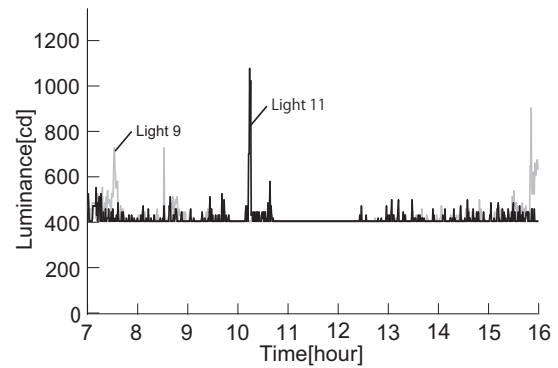


Fig. 10. History of the luminous intensity distant from target points

To verify the validity of the proposed method, an operational experiment was conducted in an environment simulating a real office. The experiment demonstrated that the method we propose can realize desired illuminance levels with energy efficient lighting patterns by estimating daylight from windows even when illuminance sensor positions do not coincide with the points where users wish to realize desired illuminance levels. These results demonstrate that individualized illuminance environments can be realized even where illuminance sensors cannot be placed on workplanes.

REFERENCES

- [1] H. Yoshida and T. Mannam :Technologies — Panasonic Electric Works — Panasonic - "Residential Lighting Fixture with Automatic Brightness Adjustment Function by Sensor", panasonic, Technical Report Vol. 57, No. 4, 2009
<http://panasonic-electric-works.net/technologies/report/574e/main02.html>
- [2] P. Boyce and N. Eklund and N. Simpson : "individual Lighting Control Task Performance, Mood, and Illuminance", J. of the Illuminating Engineering Society, pp.131-142, 2000.
- [3] M. Miki, T. Hiroyasu, and K.Imazato : "Proposal for an intelligent lighting system, and verification of control method effectiveness", Proc. IEEE CIS, pp.520-525, 2004.
- [4] M. Miki, K. Imazato, and M. Yonezawa : "Intelligent lighting control using correlation coefficient between luminance and illuminance", Proc. IASTED Intelligent Systems and Control, vol.497, no.078, pp.31-36, 2005.
- [5] M.Miki, Y.Kasahara,T.Hiroyasu and M.Yoshimi : "Construction of Illuminance Distribution Measurement System and Evaluation of Illuminance Convergence in Intelligent Lighting System", IEEE Sensors2010 Acoustic and Optical Sensing Systems, 2010.
- [6] S.Inoue, MITSUBISHI ESATE COMPANY Ltd.:Towards "the City of the Future"
http://www.jetro.org/documents/green_innov/Shigeru_Inoue_Presentation.pdf
- [7] S. Tanaka, M. Miki, T.Hiroyasu and M.Yoshikata : "An evolutionary optimization algorithm to provide individual illuminance in workplaces",Proc:Systems Man and Cybernetics, 2009. SMC 2009. IEEE International Conference on ", pp.941-947, 2009.
- [8] Y. Ohtani, M. Ohkawa, A. Uchida and T. Yamaya : "Illuminance Calculation Using Monte Carlo Method", J. Light &Vis. Env., Vol. 24, No.1, pp.1_42-1_49, 2000.
- [9] C. Reinhart and S. Herkel : "The simulation of annual daylight illuminance distributions — a state-of-the-art comparison of six RADIANCE-based methods", Trans. Energy and Buildings, Volume 32, Issue 2, pp.167-187,2000.
- [10] D. Li, G. Cheung and C. Lau : "A simplified procedure for determining indoor daylight illuminance using daylight coefficient concept", Trans. Building and Environment Volume 41, Issue 5,pp. 578-589, 2006.

Event Detection using Archived Smart House Sensor Data obtained using Symbolic Aggregate Approximation

Ayaka ONISHI¹, and Chiemi WATANABE²

^{1,2} Graduate School of Humanities and Sciences, Ochanomizu University, Bunkyo-Ku, Tokyo, Japan

Abstract - In recent years, the widespread usage of network and sensor technologies has resulted in an increased number of applications that use various types of sensors. We propose the development of a futuristic house that can adapt to rapid changes in information technology. This project is called "The Ocha House Project." We focused on data obtained using sensors and proposed and implemented a database system, which stores information pertaining to critical parameters in the house. This system can form the basis for developing other applications. We used the symbolic aggregate approximation (SAX) to quickly retrieve data. SAX converts data from a time series into a string. In addition, we created an index by applying suffix trees. In this study, we investigated queries that can be used for detecting events in a smart house using the SAX index and archived sensor data.

Keywords: Symbolic aggregate approximation, event detection; sensor data

1 Introduction

Owing to the widespread use of networks and sensors technology in recent years, significant opportunities for using sensors have emerged places such as offices, and shops.

In this study, we focus on sensor applications in daily life. We are currently participating in the "Ocha House Project" which aims to develop a futuristic house that would facilitate a life style using a variety of IT technologies [1]. The methods and systems proposed in the project will be implemented in an experimental smart house called "Ocha House" (Figure 1) [2].



Figure 1. Experimental smart house called "Ocha House."

In our study, we focus on technologies involving the storage of sensor data in a manner that enables the system to efficiently answer queries arising from using sensor applications. By querying sensor data, applications can extract "house's information." This includes information about events that occur in the house, such as the time of opening or closing a particular door, the duration for which the light in the living room is used, among others. If such information can be retrieved, many useful devices can then be implemented using the data. Some examples include tools that manage who may enter or exit a room and those that calculate the relationship between the behavior of occupants in the house for a given time period and the corresponding electricity bill.

To extract such data, sensor data needs to be archived into a database system. However, because sensors continuously measure parameters and transmit data, sizeable amounts of data will need to be stored on the database. To efficiently retrieve and analyze this data, we need to provide a methodology that can respond to queries for such large volumes of data.

Thus, we propose a methodology that can effectively retrieve data on house parameters from the large dataset that has been archived. First, we leverage the symbolic aggregate approximation (SAX) index structure [3] proposed by Lin et. al.

The purpose of SAX is to quantize the original sensor data, which is usually a sequence of numerical values, and translate them to character strings. In this paper, we name the character strings according to the sensor data as "SAX index".

We assume that queries are issued using the query by example (QBE) style. Therefore, sequences of numerical values are issued as a query, and the system should then answer several numerical sequences, which are similar to the query pattern. Our system translates query sequences into the corresponding character string and retrieves substrings of SAX indices that are similar to the query string. One of the features of the SAX index is that various techniques dealing with character strings can be utilized for query retrieval and data mining. For example, motif discovery [4] using SAX can be implemented.

Because the sensor continually measures data and the sensor data sequence also increases in length, the corresponding SAX index will be a long string. Therefore, we apply the suffix tree as a method for effectively retrieving substrings from a long SAX index. However, we should note

that the suffix tree index is usually used for exact matches. For event detection such as the opening and closing of doors, the system needs to perform similarity retrieval because sequences assigned to identical events have slightly different patterns depending on the person(s) triggering the event.

As a similarity measure, we then propose the application of similarity retrieval according to the edit distance. We define an editing cost for the edit distance based on the definition of the SAX approximation distance and are able to realize the fast similarity retrieval using the error-tolerant recognition algorithm [5], which deals with the edit distance retrieval method with a tree.

In Chapter 2, we explain the SAX method. Chapter 3 describes the application of the suffix tree to the SAX index, while chapter 4 presents the application of an index to the stored sensor data. In chapter 5, we describe the similar retrieval method using the edit distance, and Chapter 6 describes the investigations of event detections. Chapter 7 briefly discusses related research, while the conclusions and challenge faced are mentioned in chapter 8.

2 SAX

SAX is an expression technique used for time series data, and it quantizes this data into a string. Two parameters are required for generating a SAX index: length of the string w (or the amount of data for converting one alphabet) and the number of alphabetic types a . These values are experimentally determined on the basis of the data. The steps for converting time series data into a string are as follows: (Figure 2 shows an example, where the dashed line is the sensor data that applies SAX.)

- (1) Data is divided into w equal sized “frames.”
- (2) The mean value of each frame is calculated.
- (3) The regions with numerical sensor data values are divided into multiple sub-regions, and the alphabetic characters a, b, and c are assigned to the corresponding sub-regions.
- (4) The mean value obtained in (2) is converted into characters according to the sub-region that is described in (3).

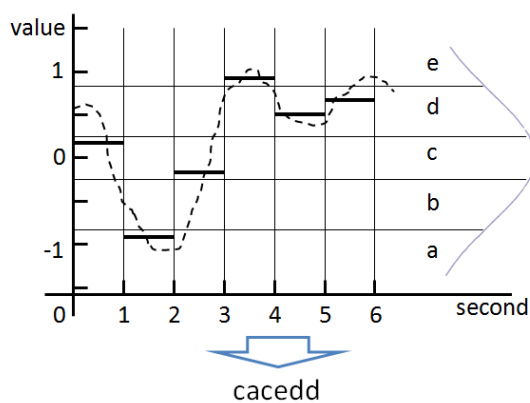


Figure 2. Time series data converted into a string by SAX.

There are three advantages of using SAX: (1) it is easy to implement, (2) it has a “symbolic” approach that allows lower bounding of the true distance, and (3) users can apply retrieval and data-mining techniques for obtaining the text string.

In this study, we implemented the methods described above and created a SAX index that converted sensor data into strings [1]. A query to the SAX index must first be converted to a string query. Time can be calculated from the position of the substring that matched the query.

We have introduced SAX as an indexing technique, and we translate sensor data into character strings.

In this system, we assume that queries are issued using the QBE style. That is, a sequence of numerical values is issued as a query, and the system responds to several sequences of numerical values that are similar to the query sequence. Our system also translates query sequences into the corresponding character string and retrieves substrings of the SAX index that are similar to the query string.

To perform a similarity, the system calculates the distance between the query sequence (Q) and subsequences of the sequence stored in the database. ($\{C_1, \dots, C_n | C_i$ refers to the subsequences of the sequence C) and answers a set of subsequences whose distances are less than a specified threshold. The Euclidean distance is often used as a measure of distance in time series data (Fig. 3 (A)).

On the other hand, the distance between a query string \hat{Q} , which is translated into a character string by SAX, and \hat{C}_i , which is one of the substrings of the SAX index, is defined as the sum of the square of the distance between $\text{dist}(\hat{q}_i, \hat{c}_i)$ (Fig. 3 (B)), where \hat{q}_i and \hat{c}_i are the i -th characters in the strings \hat{Q} and \hat{C} , respectively, w is the number of characters in the SAX index, and n is the amount of original data.

$$(A) \text{ Original data } D(Q, C) \equiv \sqrt{\sum_{i=1}^n (q_i - c_i)^2}$$

$$(B) \text{ SAX } \text{MINDIST}(\hat{Q}, \hat{C}) \equiv \sqrt{\frac{n}{w} \sum_{i=1}^w (\text{dist}(\hat{q}_i, \hat{c}_i))^2}$$

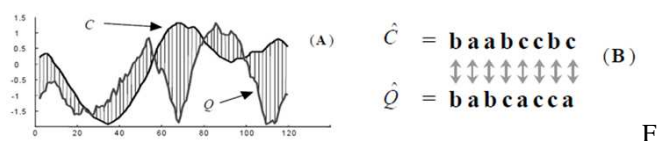


Figure 3. Definition and a schematic of distance between strings in SAX.

The distance $\text{dist}(\hat{q}_i, \hat{c}_i)$ between the characters \hat{q}_i and \hat{c}_i are defined by the distance between the maximum value assigned to the character \hat{q} and the minimum value assigned to the character \hat{c} . For example, the left side of Figure 4 shows the distribution of values in the time series data. Since the SAX index normalizes values before quantization, they follow a normal distribution. The time series values are divided into four types of characters, as shown in Figure 4. The distance $\text{dist}(\text{“a,” “d”})$ is found to be equal to 1.34 because the maximum value for “a” is -0.67 , and the minimum value for

“d” is 0.67. There is no need for the system to measure the distance between characters whenever a query is issued because a table showing the distances between characters can be prepared in advance (The table in Figure 4 shows the distance).

By defining these distances, the distance between the strings using SAX is the approximate distance for the original time series data and is guaranteed to allow lower bounding of the true distance.

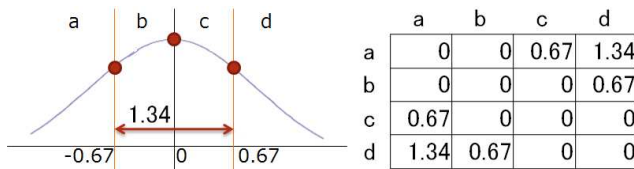


Figure 4. Table and illustration of the distance between the characters in SAX.

3 Application of suffix tree to SAX index

Queries using the SAX index retrieve substrings of the SAX index, which are similar to the query string. However, when the SAX index becomes a long sequence, it is not efficient to retrieve subsequences by sequential scanning. We therefore apply a suffix tree to the SAX index as a data structure that allows fast string retrieval.

The suffix tree is a data structure that represents suffixes of a given string as a tree. Each suffix corresponds to a different path in the tree. A leaf node takes the position of the corresponding suffix in the original string. For example, Figure 5 shows the suffix tree construction for the suffixes in the SAX index “babac.” The suffix of the string “babac” is five: “babac,” “abac,” “bac,” “ac,” and “c.” When users want to retrieve data, they will traverse a suffix tree beginning at the root. For example, if a user retrieves “ac” on the tree in Figure 5, he/she arrives at the leaf node 3. Therefore, he/she will know that “ac” exists in the third position from the beginning of the original string.

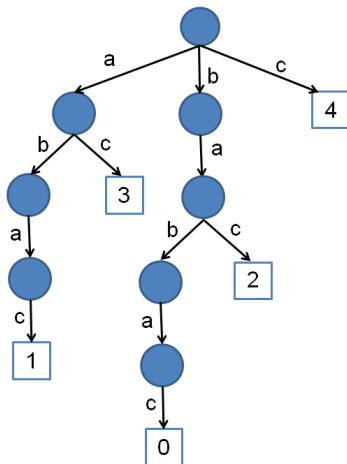


Figure 5. Suffix tree construction for the string “babac.”

To confirm the effectiveness of the suffix tree, we compared the retrieval speed using the SAX index for data that had been archived over a two-day period and corresponded to 2,230,000 characters. Table 1 shows the results obtained, and they confirm that the SAX index that used the suffix tree retrieves data faster than a sequential scan.

Table 1. Comparison of speed of retrieval.

suffix tree	No suffix tree (Sequential Scan)
3ms	267ms

4 Applying indices to sensor data

Data are sent continuously by the sensor, and the end point of the sensor data sequence is always changing. Therefore, we should note the following two problems that may be encountered when the SAX index and suffix tree are generated for the sensor data.

- (1) The sensor data should be normalized before generating the SAX index. However, we need to use the data’s statistics for the normalization process. It is difficult to normalize the data correctly because sensor data are continuously being accumulated.
- (2) The suffix tree is a tree of the suffix pattern. Since suffixes change every time new sensor data arrives, the depth of the tree increases exponentially.

To solve the first problem, we currently record the statistics when the accumulated data has reached an appropriate length. We currently set up the accumulated data for one day. Sensor data is then normalized using these statistics. However, this approach is a temporary solution, and it is necessary for an alternative solution to be found. If the distribution of sensor data changes significantly after recording the statistics, there is no guarantee that the lower bounding of the approximate distance is allowed because the values assigned to letters are based on the normal distribution.

In order to resolve the second problem, we specified a limit for the depth of the suffix tree. This approach is similar to the use of sliding windows for retrieving subsequences of time series data. In SAX [1], the same approach for dimensionality reduction has been applied; hence, our approach is appropriate, except for the case where very long events are to be detected.

5 Preliminary investigations for event detection

In this section, we consider whether events that are categorized as “house information” can be detected using the SAX index, which applied a suffix tree. For example, the event “opening and closing a door” generates several

sequence patterns. The speed of the moving door depends on the person who opens the door and/or the situation in which the door is opened. We need to examine whether such patterns can be returned by a query using the SAX index and the corresponding suffix tree index. However, we are already aware of two problems that exist when detecting events by query processing using a SAX index and suffix tree.

The first problem is that normal suffix trees cannot deal with similarity matches. SAX can perform a similarity match using an approximate distance to compare two strings. However, the suffix tree was applied to quickly retrieve substrings from a long SAX index. Because the suffix tree is essentially a data structure for substrings that match exactly, a similarity match cannot be performed. For example, Figure 6 shows the sequence data for the door sensor. From the sequence, it is observed that there are three behavior patterns. Subsequence (A) and (B) show the sensor value behavior when the same person opens and closes the door twice, while subsequence (C) indicates that the person who opened the door kept it open for a longer time period, after which it was closed. The bottom of Figure 6 shows the SAX index ($w = 53, a = 8$) that was converted into string values. For visualization purposes, the SAX index is compressed using run length encoding. If the retrieved query is a time sequence (Q), (A), (B), and (C) are not returned as the result because neither of them match the string (Q) exactly.

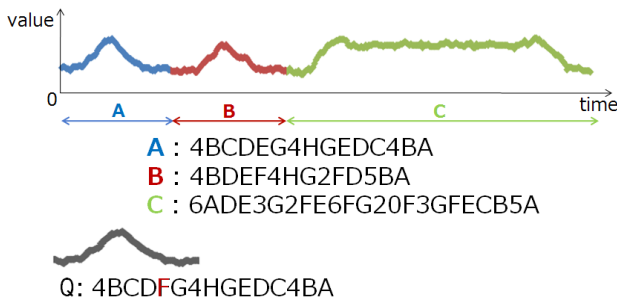


Figure 6. Behavior of the actual sensor data during opening and closing of door.

The second problem concerns the inability to determine the difference in speed when the door is being opened and closed. For example, Figure 7 shows the variation in the sensor data sequence as someone first opens and closes the door at normal speed, after which he/she slowly opens and closes the door. The bottom of Figure 7 shows the SAX index ($w = 41, a = 8$) that was converted into string values.

The time series A and D represent the same event (“opening and closing the door”), but they have a slightly different SAX index owing to the difference in the speeds with which the actions were performed. Currently, we cannot differentiate between these values.

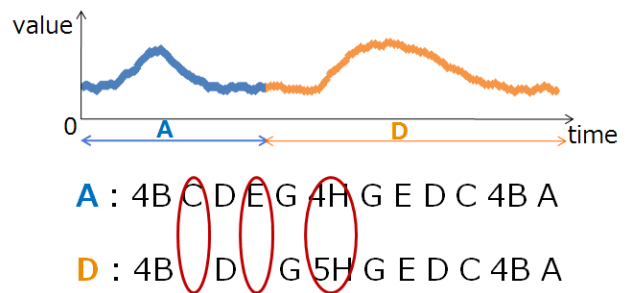


Figure 7. Difference in opening and closing time of door.

6 Similar retrieval method using the edit distance

In order to facilitate situations such as those mentioned in section 5, we define the edit distance as the distance between the strings.

The edit distance is the minimum total value of the costs (replace, delete, insert) required for converting a string Q into a string C. The minimum total value is considered to be the distances $dist(Q,C)$ between the string C and the string Q. For example, if the string Q is “AFC” and the string C is “ABCE,” strings Q and C will match if the second character of Q (character “F”) is replaced by the character “B” and if the character “E” is inserted into the fourth character of Q. If the cost is defined as the number of “replace” and “insert” operations that are required, then, $dist(Q,C)$ will be 2.

An approach that can be used to retrieve a partially similar string using a suffix tree and applying the edit distance has been proposed as the error-tolerant recognition algorithm [3]. This can be adopted to perform similar retrieval of strings using a suffix tree.

The second problem described in the previous section may be solved using the dynamic time warping (DTW) distance.

DTW is a measure of the distance of the time series. It can be calculated using the distance absorbed scaling and the time gap. Chen [5] describes the application of DTW to the cost value of the edit distance. We can therefore calculate the edit distance based on DTW by applying this approach.

In section 6.1, we describe the edit distance in the SAX index, and in section 6.2, we describe the experiments involving the detection of the “doors” event.

6.1 Definition of edit distance in SAX index

According to DTW, the editing costs of the editing distance comprise.

- Replacement Cost

The replacement cost is determined by the distance $dist(a, b)$ between characters in SAX (section 2).

- Gap (insert and delete) Costs

The insert cost is defined as the distance between the inserted character x and the character y, which is before one of the positions into which the insertion is made. For

example, to compare the costs of “ABC” and “AC,” the insert cost associated with inserting B between A and C in the second string is $\text{dist}(A,B)$. The same concept applies to the delete cost, which is defined as the distance between the delete character x and the character y, which is before one of the positions that is to be deleted.

Based on this concept, the distance of a string from the first character to the respective i-th and j-th characters of Q and C in the SAX index can be defined as follows:

$$P(Q_i, C_j) = \min \left\{ \begin{array}{ll} P(Q_{i-1}, C_{j-1}) + \text{dist}(q_i, c_j), & \# \text{permute} \\ P(Q_{i-1}, C_j) + \text{dist}(q_{i-1}, c_j), & \# \text{delete} \\ P(Q_i, C_{j-1}) + \text{dist}(q_i, c_{j-1}) \} & \# \text{insert} \end{array} \right.$$

However, in our case, we anticipate that user requests will include the desire to control the speed with which a door is opened or closed. Therefore, in order to consider speed, the retrieval uses the following equation:

$$P(Q_i, C_j) = \min \left\{ \begin{array}{ll} P(Q_{i-1}, C_{j-1}) + \text{dist}(q_i, c_j), & \# \text{permute} \\ P(Q_{i-1}, C_j) + \alpha * \text{dist}(q_{i-1}, c_j), & \# \text{delete} \\ P(Q_i, C_{j-1}) + \alpha * \text{dist}(q_i, c_{j-1}) \} & \# \text{insert} \end{array} \right.$$

A control parameter α can be specified by a user on delete and insert cost. If α is increased, the retrieval can include speed information for the “doors” event.

6.2 Experiments involving “doors” event detection

We performed “doors” event detection experiments using the edit cost as it was defined in the previous section. Figure 8 shows SAX strings of the experimental data and a query Q. The seven events that are observed include four successive open-and-close events, followed by a slow open-and-close event and a fast open-and-close event, and finally, a case where a subject kept the door open for a longer time before closing it. For this example, we performed detection experiments during the time period shown in red as the query Q.

Figure 9 shows the results of detection experiments that resulted in changed thresholds, delete, and insert costs. The orange parts are substrings of the detection result.



Figure 8. Behavior of experimental data ($w=287, a=10$)

The threshold in case (1) is set to 0.2 and $\alpha = 1$, while the threshold in case (2) is set to 0.2 and $\alpha = 10$.

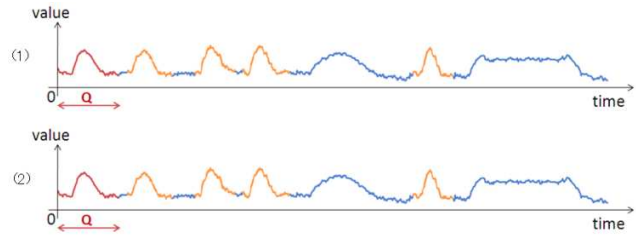


Figure 9. Detection results

In case (1), the first three events and the faster event could be viewed as detection results. However, when only α is varied and the threshold is not varied as in case (2), neither the fifth event nor the last event were detected. We had expected that all events would have been detected in case (2), but the result in case (2) were the same as in case (1), the parameter α did not function correctly. One possible reason may be that it is too small to affect the delete cost. Therefore, we will not regulate the collective costs to insert and delete, but will instead separately regulate them.

7 Related research

SAX is based on the piecewise aggregate approximation [6], which quantizes time series data using a fixed interval; it is an extended approach that allows the lower bounding of the true distance in the approximate distance. It can be applied to a variety of search technologies and techniques for mining strings.

In this paper, we performed fast retrieval of partial time series data using suffix trees and edit distances. Chen also performed a similar retrieval for the time series data using edit distance [7][8]. In [7], the original value was not quantized, and the cost was determined on the basis of whether the distance between the two characters in the time series was within a specified threshold. In [8], a cost is defined that satisfies the axioms of distance and speeds up the retrieval using the measured index.

In addition, iSAX is an extension of SAX and is applicable for very long time series [9]. This approach generates an index for short SAX strings that are split using a sliding window and has achieved a fast retrieval for substrings of a long string. Unlike the suffix tree that we adopted, the iSAX index considers n-dimensional data as a string of n characters that are divided using a sliding window and which generate an index on the feature space using an n-dimensional vector. However, this approach does not correspond to the scaling of time series.

Gao et. al. [10], Zhi et. al. [11], and others have conducted research into part similarity retrieval of streaming data. Sakurai et. al have studied a similar retrieval method using high-speed DTW in streaming data [12].

We have tried a similar retrieval method using a speed-up approach of the string retrieval method for time series data that is quantized by SAX. In future, the effectiveness of these methods needs to be verified.

8 Conclusions

In this paper, we implemented a SAX index that allows fast retrieval of archived sensor data for a smart house. In addition, we achieved fast similarity event detections using a suffix tree and performed experiments for event detections.

In the future, we will focus on improving the proposed retrieval method and perform additional experiments using different events.

9 References

- [1] Ocha House:
<http://www.siiio.jp/index.php?OchaHouse>
- [2] Ocha House Projects:
<http://www.siiio.jp/index.php?OchaHouseProjects>
- [3] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. "A Symbolic Representation of Time Series, with Implications for Streaming Algorithms," SIGMOD workshop, 2003.
- [4] Yoshiki Tanaka and Kuniaki Uehara. "Motif Discovery Algorithm from Motion Data,"; 18th Annual Conference of the Japanese Society for Artificial Intelligence (JSAI), June 2004.
- [5] Kemal Oflazer. "Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction,"; Journal of Computational Linguistics, Vol. 22, Issue 1, March 1996.
- [6] Yi, B, K, and Faloutsos, C. "Fast Time Sequence Indexing for Arbitrary Lp Norms,"; 26st International Conference on Very Large Databases, pp 385–pp 394, 2000.
- [7] Chen L. Chen, M.T.zsu, and V. Oria. "Robust and efficient similarity search for moving object trajectories,"; CS Tech. Report. CS-2003-30, School of Computer Science, University of Waterloo.
- [8] Lei Chen and Raymond Ng. "On the marriage of Lp-norms and edit distance,"; 30th International Conference on Very Large Data Bases (VLDB 2004), 2004.
- [9] Jin Shieh and Eamonn Keogh. "iSAX: Indexing and mining terabyte sized time series,"; 14th ACM SIGKDD international conference on Knowledge discovery and data mining, 2008.
- [10] Like Gao and Sean. X. Wang. "Continually Evaluating Similarity-Based Pattern Queries on a Streaming Time Series,"; The 2002 ACM SIGMOD International Conference on Management of Data, pp 370–pp 381, June 2002.
- [11] Y. Zhu and D. Shasha. "StatStream: Statistical monitoring of thousands of data streams in real time,"; 28th International Conference on Very Large Data Bases (VLDB 2002), pp 358–pp 369, August 2002.
- [12] Y. Sakurai, C. Faloutsos, and M. Yamamuro. "Stream Monitoring under the Time Warping Distance,"; 23rd IEEE International Conference on Data Engineering (ICDE 2007), pp 1046–pp 1055, April 2007.

Semi-ShuffledBF: Performance Improvement of a Privacy-Preserving Query Method for a DaaS Model Using a Bloom filter

Shizuka Kaneko¹, Chiemi Watanabe², and Toshiyuki Amagasa³

^{1,2} Ochanomizu University Graduate School of Humanities and Sciences, 2-1-1 Otsuka, Bunkyo-ku, Tokyo, Japan

³ Tsukuba University Systems and Information Engineering, 1-1-1 Tennodai, Tsukuba-shi, Ibaraki, Japan

Abstract - In database-as-a-service, users can utilize a database service that is maintained by third parties via the Internet. In such an environment, it becomes difficult for the user to hide confidential information from the data administrator. To solve this problem, we previously proposed "Privacy-Preserving Query Method Hiding Schema Information Using a Bloom filter." With this method (ShuffledBF), we generate a Bloom filter for the queries of each tuple and shuffle bit sequence by using the key in each tuple. In this way, it is possible to prevent the leakage of bit patterns. On the other hand, it is problematic that the time required is proportional to the number of tuples processed, because we must restore the shuffled bit sequence by applying a hash function to each tuple when we run the query. While, in contrast, the case of a Non-Shuffled Bloom filter (Non-ShuffledBF) has a security problem. Therefore, we propose a hybrid technique called Semi-ShuffledBF that consists of two steps: 1) Non-ShuffledBF and 2) ShuffledBF.

Keywords: database outsourcing, cloud computing, database-as-a-service(DaaS), Bloom filter, Privacy-Preserving Query Method

1 Introduction

Recently, database-as-a-service (DaaS) has attracted considerable attention. DaaS provides a data management service in the cloud computing environment. Many DaaS services have already been provided by Amazon, Google, Microsoft, et al., such as S3, EC2, SimpleDB, Azure, Google Apps Engine, and so on. DaaS services are used by individuals and small companies who find it difficult to administer the DBMS on a constant basis.

In such an environment, we should note that DaaS administrators are third parties from the viewpoint of DaaS users. Therefore, it is natural that users need to hide sensitive data from DaaS administrators. To achieve such a user requirement, techniques for privacy-preserving query processing have been investigated by many researchers [1][2][3][5][7][9]. By using the investigated techniques, users can store data that is encrypted at the client end, and can issue queries to the encrypted database to receive the appropriate

results, without leaking the original value of the stored data in the DBMS.

Figure 1 shows the general flow of the Privacy Protection Method.

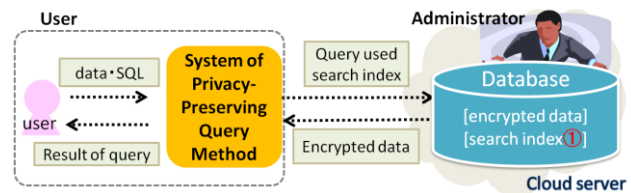


Figure 1: System of Privacy Preserving Query Method.

As the first step, the system encrypts each tuple on the client side, and sends the encrypted tuples to the database server. During this time, the system also sends the search index (① in Figure 1) for the corresponding tuple. The search index is used by the query processor on the database server to process queries without leaking the original value of the encrypted tuples. Previous studies [1][2][3][5][7][9] prepared the search index for each attribute in each tuple, and a scheme for making an index has been proposed according to the data types and operation types issued in the query.

In our previous studies [12], we have proposed ShuffledBF, which is a technique for privacy-preserving processing using a Bloom filter. ShuffledBF combines the search indices for multiple attribute values in a tuple, and then, it conceals the schema information and distribution information of the original table. ShuffledBF generates hash values that are used for generating a Bloom filter by using the attribute and the identifiable values of the tuple. Therefore, even if two tuples have the same attribute value, the hash values that are generated from the attribute values are different from each other. By using this mechanism, ShuffledBF guarantees a high level of security. However, ShuffledBF has the problem of low processing speed.

In this study, we focus on Non-ShuffledBF. Non-ShuffledBF generates hash values based only on the attribute value, and if two tuples have the same attribute value, then the hash values are the same. The performance of query processing is

obviously better by using Non-ShuffledBF than by using ShuffledBF; however, it cannot guarantee privacy preserving against adversaries. In this paper, we propose Semi-ShuffledBF, which is combining ShuffledBF to utilize the advantages of both the mechanisms. We investigate the query processing time in a single-server environment, and find that Semi-ShuffledBF can improve the query processing time in circumstances when the selection ratio is low.

The paper is organized as follows. The previous study on ShuffledBF is presented in Section 2, and the proposed structure of Semi-ShuffledBF is described in Section 3. We show the results of our performance measurements in Section 4 and the related research in Section 5. Finally, we discuss our conclusions and future work in Section 6.

2 ShuffledBF

ShuffledBF is a Privacy-Preserving Query Method using a Bloom filter. ShuffledBF can carry out a selection protection operation to a single relation; when selecting, it can carry out the operations of exact string, matching part, and number attribute. Therefore, ShuffledBF has high security, and its schema information and value cannot be guessed from the original data of the query and the stored data on the server.

The generation of the search index is described in Section 2.1, and the query is described in Section 2.2. The means of transrating a numeric attribute is presented in Section 2.3.

2.1 Generating the Search Index

In this section, we describe the means to convert tables, encrypt tuples, and convert queries. This method uses a Bloom filter for the search index. A Bloom filter is an index that can quickly determine whether a collection contains an element. It is characterized as space efficient and can perform faster searches and OR calculations, and detect false positives. Figure 2 shows an example of a conversion table.

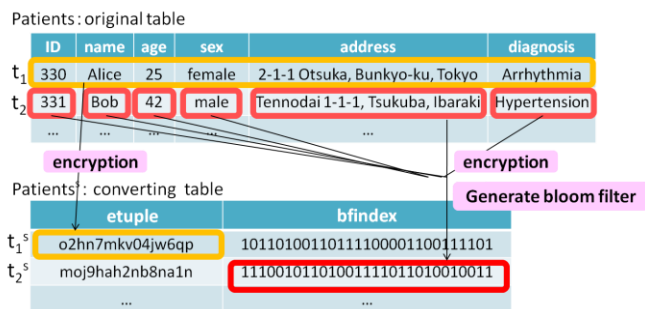


Figure 2: Example of a conversion table.

The table Patients is composed of the attributes ID, name, sex, address, and diagnosis. We prepare Patients^s in the server that correspond to Patients. This table has only two attributes: etuple and bfindex. The attribute etuple stores the values of the encrypted tuples. The attribute bfindex is the search index of the tuple. Because there is only one bfindex created per

tuple regardless of the source schema, it is difficult to determine the attributes of Patients that are derived from Patients^s. This makes the distribution of the values confusing and prevents attacks on the data.

Figure 3 shows a flow diagram of generating a Bloom filter index (ShuffledBF) from the tuple t1.

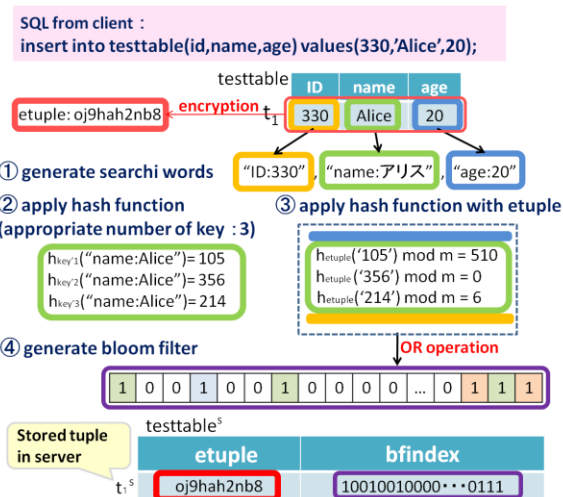


Figure 3: Generating a Bloom filter index.

Bfindex is an index that uses a Bloom filter. Its structure is based on the attribute names and values of the tuple. For example, the corresponding word of the value “Alice” of the attribute “attribute” in Figure 3’s tuple t1 is “name:Alice”(① in Figure 3). For each word that is made in this way, multiple hash functions are applied.② in Figure 3 is an example of applying three hash functions for “word:Alice.” We used the HMAC hash function and some required keys. ② in Figure 3 uses three keys: key1, key2, and key3. Next, we apply these hash values via HMAC using etuple as a key. If the tuples have the same value of standing bits in different locations, applying a second hash function prevents the gathering of features of the original data from the bit pattern. A Bloom filter index that does not apply the first hash function is called Non-ShuffledBF (NSBF).

2.2 Query

In this section, we describe the means of transrating a query. Figure 4 shows an example of transrating a query.

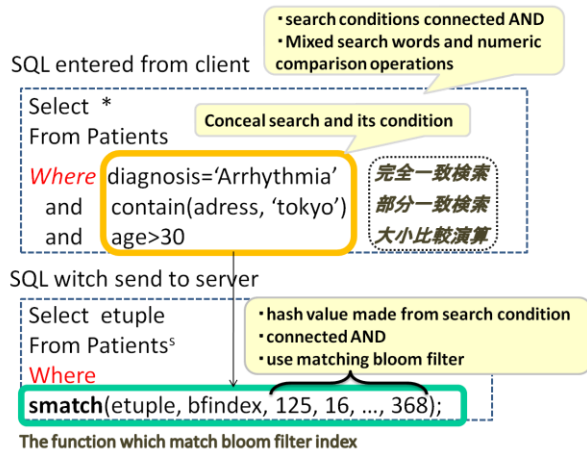


Figure 4: Example of transration of query.

The upper SQL in Figure 4 is entered by the client, and the lower SQL is sent to the server. The lower SQL replaces the condition of the attributes and the text search of each tuple to the conditions of bfindex. Therefore, database administrators cannot read what we have specified as the condition of the attribute.

Next, Figure 5 shows how query processing proceeds on the server.

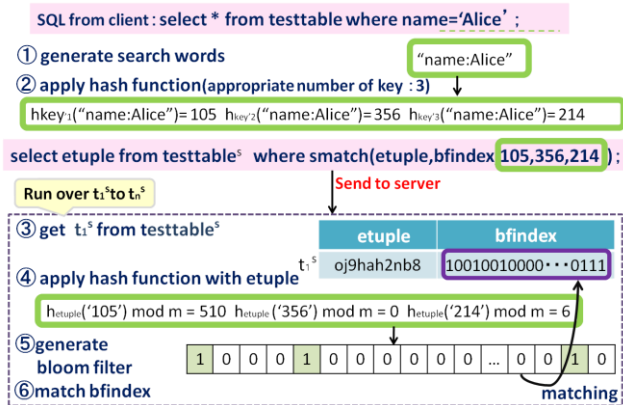


Figure 5: Example of processing query.

First, generate the words from the search condition (① in Figure 5), apply the first hash function, and then, send the SQL to the server (② in Figure 5). At the server side, match the query conditions by processing smatch function written in where section in SQL by each tuple. In the smatch function, we apply the hash function used by the key as the hash values generated at the client side (105,356,214 in ④ in Figure 5). We match the values to the Bloom filter (⑤ in Figure 5). In this way, the user can hide the type of the number and operation to the schema information as well as the search condition.

2.3 Transrating a Numeric Attribute

Because it is impossible for the Bloom filter to compare numbers, we need to transrate the numbers in the data into words in order to apply a Bloom filter to the numbers. Basically, the domain is divided into several buckets of numeric attributes, and the generated words are added to the bucket name and attribute name.

ID	name	age	sex	address	diagnosis	
t ₁	330	Alice	25	female	2-1-1 Otsuka, Bunkyo-ku, Tokyo	Arrhythmia

Divide and substitute 25: {eq:B3,lt:B1,lt:B2,mt:B4,mt:B5,mt:B6,mt:B7,mt:B8,mt:B9,mt:Ba,mt:Bb}

	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	Ba	Bb
25	lt:B0	lt:B1	lt:B2	eq:B3	mt:B4	mt:B5	mt:B6	mt:B7	mt:B8	mt:B9	mt:Ba	mt:Bb
55	lt:B0	lt:B1	lt:B2	lt:B3	lt:B4	em:B5	mt:B6	mt:B7	mt:B8	mt:B9	mt:Ba	mt:Bb
88	lt:B0	lt:B1	lt:B2	lt:B3	lt:B4	lt:B5	lt:B6	lt:B7	lt:B8	em:B9	mt:Ba	mt:Bb

Figure 6: Representation of bucket of numeric attributes.

In Figure 6, 25,55,88 represent each set of words.

If the limit of the bucket is less than the value of v against all buckets $B = B1, \dots, Bb$, we add the word 「<attribute name>:lt:<bucket name>」. If the limit of the bucket is more than the value of v, we add the word 「<attribute name>:mt:<bucket name>」. In another case, we add the word 「<attribute name>:eq:<bucket name>」. When comparing the values with the magnitude using this method, for example, if you want to get a value greater than 75, you focus on (B7) (left of (B8)) that contain 75, and search for the tuple that has the word 「<attribute name>:lt:B7」. On the other hand, if you want to get a value less than 35, you focus on the right bucket (B5) and search for the tuple that has the word 「<attribute name>:lt:B5」. Thus, we can treat a numeric comparison operation as a matching string.

3 Semi-ShuffledBF

ShuffledBF is secure, because it is difficult to estimate the original data from the Bloom filter on the server. However, there is a problem that the processing time is very long, because we apply the hash function against all tuples with each query (see Figure 10). Therefore, we propose Semi-ShuffledBF, which can perform privacy-preserving queries without rack safety and is faster than combined Non-ShuffledBF that does not apply a conversion function.

We describe the basic idea of Semi-ShuffledBF in Section 3.1, the way of inserting data in Section 3.2, and the method of the query in Section 3.3.

3.1 Basic Idea

Semi-ShuffledBF is an index that is a combination of the ShuffledBF and Non-ShuffledBF indices, which narrows down the number of tuples to be applied to a hash function. At

query time, we expect to reduce the number of tuples by applying it to the hash function by narrowing down using Non-ShuffledBF and ShuffledBF indices. We apply the following hash function on the Non-ShuffledBF index:

$$h'_i(x) = \lfloor \lfloor g_j(h_i(x) \bmod \lfloor m/l \rfloor) \rfloor \rfloor \bmod m \quad \dots \quad (1)$$

Here, m is the bit length of the Bloom filter, l is an integer parameter and takes the value of at least 2, and the function g_j is set for each attribute A_j in the source table $R(A_1, \dots, A_n)$. On the other hand, on ShuffledBF we set up the remainder of m , which is the bit length of the Bloom filter, and on Non-ShuffledBF, we set up the remainder of $\lfloor m/l \rfloor$. We increase the false positives from Non-ShuffledBF by increasing the value of l , so it is difficult to estimate the original data. The g_j function is a function for determining the location of the bit standing of each attribute A_j , and for preventing the duplication of the position of a bit in a single Bloom filter. In addition, Semi-ShuffledBF does not separate Non-ShuffledBF from ShuffledBF, so the false positives may be higher by combining them. However, we consider that this can be adjusted by making the bit length m of the Bloom filter slightly longer.

3.2 Inserting data

The insertion of data into Semi-ShuffledBF is divided into the following four stages:

- (1) Generate ShuffledBF (Figure 3 in Section 2.1).
- (2) Generate Non-ShuffledBF.
- (3) Carry out an OR operation in ShuffledBF and Non-ShuffledBF.
- (4) Store the result in bfindex.

First, we generate the search words in ShuffledBF. And we generate it in Non-ShuffledBF by applying the Equation(1). We store the result by performing an OR operation in Non-ShuffledBF and ShuffledBF as Semi-ShuffledBF in bfindex.

3.3 Method of the Query

The method of the query of Semi-ShuffledBF is divided into the following two stages.

- (1) Search with Non-ShuffledBF.
- (2) Search with ShuffledBF.

First, apply Non-ShuffledBF to each tuple. Thus, only the tuples that correspond to Non-ShuffledBF are applied to ShuffledBF.

3.4 Effect of Semi-ShuffledBF

The effect of using Semi-ShuffledBF is a secure, more rapid search. ShuffledBF has the problem that the processing time is very long, because we apply the hash function against all tuples with each query. Semi-ShuffledBF uses Non-ShuffledBF, which can process rapidly and apply only the tuples matched by the Non-ShuffledBF hash function. In this way, Semi-ShuffledBF becomes faster.

Figures 7 and 8 show examples of ShuffledBF and Semi-ShuffledBF queries.

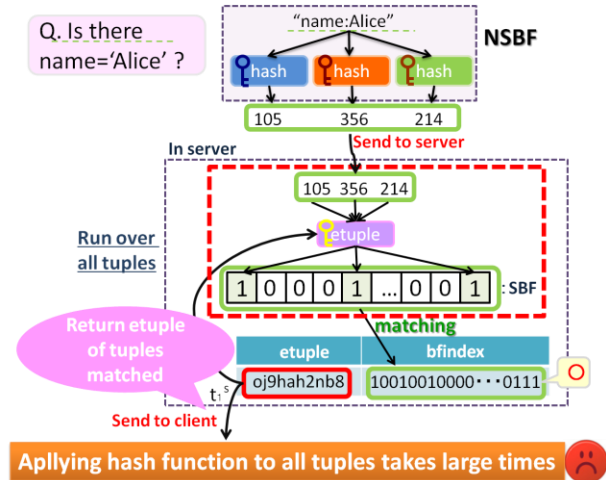


Figure 7: Example of ShuffledBF query.

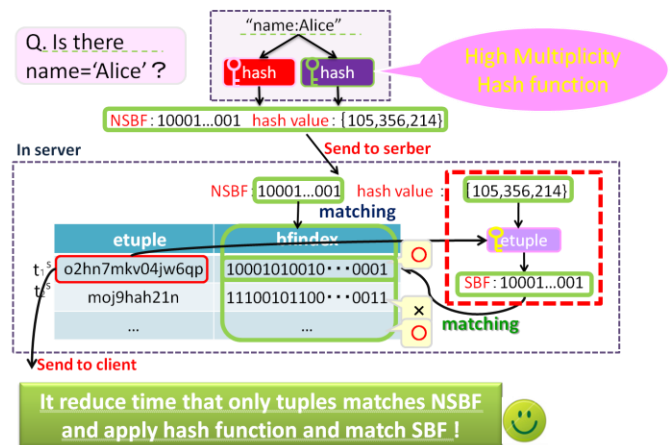


Figure 8: Example of Semi-ShuffledBF query.

The process marked within red dots in Figures 7 and 8 is the part of the process that is performed faster. Figure 7 shows the process for all tuples; on the other hand, Figure 8 shows the process for only the tuples matched with Non-ShuffledBF.

This improved processing speed is because the process for the tuples was omitted, which is not correct.

4 Performance Evaluation

As a preliminary experiment for the proposed method, we extended the programs of the Privacy-Preserving Query Method that is built on previous research, used DBMS to improve the performance, and evaluated the performance.

4.1 Experimental Environment

We evaluated the performance using a Linux Server (CPU: Intel (R) Xeon (R) 2.00 GHz Memory: 8 GB) and a database server (PostgreSQL) as the experimental environment. We used 100,000 tuples of artificial data and specified the length of the Bloom filter m as 128 bytes, the function g_j as a primary function, and the partition number l as 10.

In this study, we measured only the query processing time on the server.

In the Privacy-Preserving Query Method, users can obtain the result by re-querying the data after leeching and decoding the correct tuple that is searched on the server. In fact, the time for leeching to the client and decoding may be very long, but in this study, we do not consider this duration because it is beyond the scope of this proposed method.

4.2 Experimental Results

Following are the performance results of ShuffledBF, Non-ShuffledBF, and Semi-ShuffledBF on the experimental server.

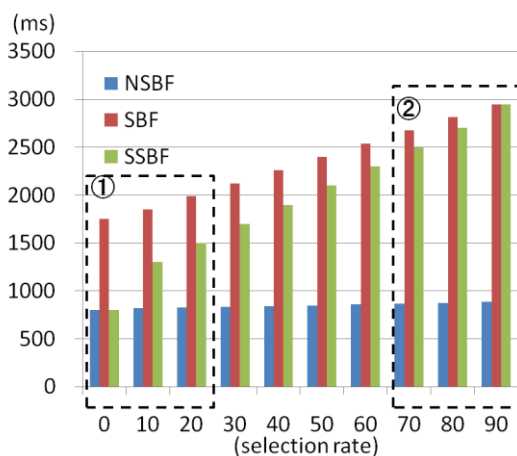


Figure 10: Performance results

(selection rate sets “the number of correct tuples/the number of all tuples”).

The case where the selection rate is a row (① in Figure 10), processing efficiency is improved when using Semi-ShuffledBF. However, the case where the selection rate is high (② in Figure 10), the processing efficiency does not improve.

4.3 Considerations

In this experiment, there are two reasons why Semi-ShuffledBF has not yet improved significantly compared to ShuffledBF.

(1) The process used for Non-ShuffledBF is not very fast.

We expected that the process would be fast, because we were basically operating with only bit, but it actually took about 700 ms. This is because you need a table scan to check all the tuples.

(2) The same action (processing times) occurs for answer tuples.

Semi-ShuffledBF reduces processes to not fit query condition. In the case that the selection rate is high, however, the effect is small because the processes are not many. Therefore, the result is the same as in the preceding section.

Following are some possible ways to improve these two issues:

(1) Grant bitmap indices for Semi-ShuffledBF.

It is suggested that bitmap indices be granted to `bindex` to make the primary search on Non-ShuffledBF faster.

Thus, because you can access the location of the bits directly, without performing a file scan of all the tuples, it is expected that the I/O costs of the disk can be reduced.

(2) Do not search by ShuffledBF.

In the case when the selection rate is high, use Semi-ShuffledBF to search most of the tuples rather than ShuffledBF.

It is suggested to calculate the selection rate after searching by Non-ShuffledBF. Thus, in the case when the selection rate is high, it is considered that all the tuples are correct and a search by ShuffledBF is not required. These approaches will increase the false positives of the search, but we believe that this problem can be solved by the artifice in client as mentioned in Section 4.1.

5 Related Research

Many studies have been performed on the Privacy-Preserving Query Method for outsourcing. Hacigumus et al. proposed to store the search index to the database on the server, how to query the index on the server generated by a user and the generation of query execution which executed divided instead of the query on client. The search index is provided for each attribute and are produced by different methods in data types and the calculation of used conditions.

In the method of generating the index, the distributed value may obtain the original value. On the other hand, Hore et al. proposed a method for splitting the bucket, which makes it difficult to estimate the value of the distribution [7]. Agrawal et al. [1] proposed a conversion method of the number attribute that preserves the relationship. This method can prevent the estimation of the original value by converting the distribution of values that are different from the original distribution. It can process both compared and combined operations. Lee et al. [8] and Hasan et al. [6] have used the proposed method. Aggregations and k-neighbor [11] used an encryption method with homomorphism, which has also been proposed by Mykletun et al. [4] and Ge et al. [10]. In these studies, there is a problem of security and performance, as exists in our research. On creating an index for each cell, there is a problem that if there are many data on the server, it is possible that someone may obtain the original value by analyzing the trend of the index values. In case of checking the condition for each tuple, there is the problem that you cannot use the index. In addition, if you create an index, someone may obtain the original value from the index. In our proposed method, the possibility of obtaining the original value is low compared to the method of generating an index for each cell, because the index is in one tuple. It is anticipated that Semi-ShuffledBF displays a performance improvement by applying a bitmap index of the Bloom filter. We conclude that the ability to obtain the original values from the index is low because it contains shuffled bits.

6 Conclusions and Future Work

We proposed Semi-ShuffledBF, which can perform Privacy-Preserving Queries without rack safety and faster than Non-ShuffledBF that does not apply the conversion function.

In the future, we plan to speed up the performance of Semi-ShuffledBF and establish its indicators of performance and security.

We will also evaluate the performance of other DaaS, such as Windows SQL Azure.

Acknowledgment

Part of this research is from “KAGAKU-KENNKYUUHI-HOJYO-KINN-WAKATE-KENNKYUU(B)(number:21700099).”

7 References

- [1] Agrawal R., Kiernan J., Srikant R., and Xu Y.: Order preserving encryption for numerical data, Proceedings of the 2004 SIGMOD International Conference, pp.563–574 (2004) .
- [2] Bellovin S. and Cheswick W. : Privacy-enhanced searches using encrypted bloom filters” (2004) .
- [3] Boneh D., Crescenzo G.D., Ostrovsky R., and Persiano G.: Public Key Encryption with Keyword Search, Proceedings of EUROCRYPT '04, vol.3027 LNCS, pp.506—522 (2004) .
- [4] E. Mykletun, G. Tsudik : Aggregation queries in the database-as-a-service model. IFIP WG 11.3 on Data and Application Security (2006) .
- [5] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. : “Executing SQL over Encrypted Data in the Database-Service-Provider Model , ”Proceeding of the ACM SIGMOD International Conference on Management of Data, pp. 216-227, (2002) .
- [6] Hasan Kadhem, Toshiyuki Amagasa, and Hiroyuki Kitagawa : “A Secure and Efficient Order Preserving Encryption Scheme for Relational Databases.” Int'l Conf. on Knowledge Management and Information Sharing (KMIS 2010) , Valencia, Spain, October 25-28 (2010) .
- [7] Hore B, Mehrotra S., and Tsudik G : A privacy-preserving index for range queries, Proceedings of the 30th International Conference on Very Large Data Bases, pp.720—731 (2004) .
- [8] S. Lee, T. Paek, D. Lee, T. Nam, and S. Kim : Chaotic Order Preserving Encryption for Efficient and Secure Queries on Databases, IEICE Transactions on Information and Systems E92.D (11) , 2207-2217 (2009) .
- [9] Ting Yu and Shushil Jajodia : Secure Data Management in Decentralized Systems, Springer-Verlag New York Inc, p.462 (2006) .
- [10] Tingjian Ge, Stanley B. Zdonik : Answering Aggregation Queries in a Secure System Model. Proceedings of VLDB 2007, pp.519-530 (2007) .
- [11] W. K. Wong, D. W. Cheung, B. Kao and N. Mamoulis : Secure kNN computation on encrypted databases, Proceedings of the 35th VLDB Conference, pp. 139-152 (2009)
- [12] Watanabe C. and Arai Y. : Privacy-Preserving Queries for a DAS model using Two-Phase Encrypted Bloomfilter, Proc. of International Conference on Database Systems for Advanced Applications (2009) .

Implementation and performance evaluation of new inverse iteration algorithm with Householder transformation in terms of the compact WY representation

Hiroyuki ISHIGAMI¹, Kinji KIMURA¹, and Yoshimasa NAKAMURA¹

¹Graduate School of Informatics, Kyoto University, Kyoto 606-8501, Japan

Abstract—A new inverse iteration algorithm that can be used to compute all the eigenvectors of a real symmetric tridiagonal matrix on parallel computers is developed. In the classical inverse iteration algorithm, the modified Gram-Schmidt orthogonalization is used, and this causes a bottleneck in parallel computing. In this paper, the use of the compact WY representation is proposed in the orthogonalization process of the inverse iteration algorithm with the Householder transformation. This change results in drastically reduced synchronization cost in parallel computing. The new algorithm is evaluated on a 32-core parallel computer, and it is shown that the algorithm is up to 7.46 times faster than the classical algorithm in computing all the eigenvectors of matrices with several thousand dimensions.

Keywords: compact WY representation, Householder transformation, inverse iteration, eigenvalue decomposition

1. Introduction

The eigenvalue decomposition of a symmetric matrix, i.e., a decomposition into a product of matrices consisting of eigenvectors and eigenvalues, is one of the most important operations in linear algebra. It is used in 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.

Such an eigenvalue decomposition algorithm involves a process of transforming a real symmetric matrix into a real symmetric tridiagonal matrix as a preconditioning step. Therefore the problem of eigenvalue decomposition can be reduced to that of a symmetric tridiagonal matrix. Several eigenvalue decomposition algorithms for a real symmetric tridiagonal matrix have been proposed. They are classified into two types. The first type of algorithm computes simultaneously all the eigenvalues and the eigenvectors. Algorithms of this type includes the QR algorithm [5] and the divide-and-conquer algorithm [8]. The second type of algorithm computes all or some eigenvalues and all or some eigenvectors. Algorithms for computing eigenvalues includes

the root-free QR algorithm [7] and the bisection algorithm [5]. Algorithms for computing eigenvectors includes the MR³ algorithm [3] and the inverse iteration algorithm [10]. LAPACK (Linear Algebra PACKage) [9], a software library for numerical linear algebra, has codes that integrate all the above-mentioned algorithms.

The inverse iteration algorithm is an algorithm for computing eigenvectors independently associated with mutually distinct eigenvalues. However, when we use the inverse iteration algorithm, we must reorthogonalize the eigenvectors if some eigenvalues are very close to each other. Adding this reorthogonalization algorithm increases the computational cost. Moreover, for this reorthogonalization, we have generally used the MGS (modified Gram-Schmidt) algorithm. However, this algorithm is sequential and inefficient for parallel computing. As a result, we are unable to maximize the performance of parallel computers. Hereinafter, we will refer to the inverse iteration algorithm with the MGS algorithm as the classical inverse iteration algorithm.

We can also orthogonalize vectors by using the Householder transformation [12], and we call this orthogonalization process the Householder orthogonalization algorithm. While the MGS algorithm is unstable in the sense that the orthogonality of the resulting vectors depends on the condition number of the symmetric tridiagonal matrix [13], the Householder algorithm is stable because its orthogonality does not depend on the condition number. The Householder algorithm is also sequential and ineffective for parallel computing, and its computational cost are higher than those of the MGS algorithm.

In 1989, the Householder orthogonalization algorithm in terms of the compact WY representation was proposed in [11]. By adopting this Householder orthogonalization, stability and effective parallelization can be achieved. Hereafter, we refer to this algorithm as the compact WY orthogonalization. In 2010, Yamamoto demonstrated the fact [13]: When this algorithm is used in the Arnoldi process, the computation time for parallel computation is less than that when the MGS algorithm is used, and the orthogonality of the eigenvectors generated using this algorithm is better than that of the eigenvectors generated using the MGS algorithm.

In this paper, we consider an implementation of the compact WY orthogonalization to the inverse iteration al-

gorithm and we evaluate its performance. Thereafter, we present a new inverse iteration algorithm for computing the eigenvectors of a real symmetric tridiagonal matrix.

The contents of this paper are as follows. In Sec.2, we explain the classical inverse iteration algorithm and describe its defect. In Sec.3, we explain the Householder orthogonalization and the compact WY orthogonalization. In Sec.4, we present a new inverse iteration algorithm, namely, the compact WY inverse iteration algorithm, whose orthogonalization process is performed by compact WY orthogonalization instead of MGS, and we explain its properties. In Sec.5, we discuss numerical experiments on parallel computers and their results. In the experiments, we compute eigenvectors of symmetric tridiagonal matrices with several thousand dimensions by using the classical algorithm and the new algorithm. It is shown that the new algorithm is up to 7.46 times faster than the classical algorithm. Section 6 presents our conclusions.

2. Classical inverse iteration algorithm and its defect

2.1 Classical inverse iteration

We consider the problem of computing eigenvectors of a real symmetric tridiagonal matrix $T \in \mathbb{R}^{n \times n}$. Let $\lambda_j \in \mathbb{R}$ be eigenvalues of T such that $\lambda_1 < \lambda_2 < \dots < \lambda_n$. Let $v_j \in \mathbb{R}^n$ be the eigenvector associated with λ_j . When $\tilde{\lambda}_j$, an approximate value of λ_j , and a starting vector $v_j^{(0)}$ are given, we can compute an eigenvectors of T . To this end, we solve the following equation iteratively:

$$(T - \tilde{\lambda}_j I) v_j^{(k)} = v_j^{(k-1)}. \quad (1)$$

Here I is the n -dimensional identity matrix. If the eigenvalues of T are mutually well-separated, the solution of $v_j^{(k)}$, Eq.(1) generically converges to the eigenvector associated with λ_j as k goes to ∞ . The above iteration method is the inverse iteration method. The computational cost of this method is of order mn when we compute m eigenvectors, and it is less than that of other methods for eigenvalue decomposition. In the implementation, we have to normalize the vectors $v_j^{(k)}$ to avoid overflow.

When some of all the eigenvalues are close together or there are clusters of eigenvalues, we have to reorthogonalize all the eigenvectors associated with such eigenvalues because they need to be orthogonal to each other. If we apply the MGS orthogonalization, the computational cost is of order m^2n . Therefore, when we calculate eigenvectors of the matrix T that has many clustered eigenvalues, the total computational cost increases significantly. In general, when we implement the inverse iteration method on computers, we use the MGS orthogonalization with the Peters-Wilkinson method [10] as the standard orthogonalization process. The MGS with the Peters-Wilkinson method is also available

on DSTEIN, the LAPACK code 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 $10^{-3}\|T\|$, we regard them as members of the same cluster of eigenvalues, and we orthogonalize all of the eigenvectors associated with these eigenvalues.

Figure 1 shows the inverse iteration algorithm based on the MGS with the Peters-Wilkinson method outlined above. We call this the classical inverse iteration method.

```

1: for  $j = 1$  to  $n$  do
2:   Generate  $v_j^{(0)}$  from random numbers.
3:    $k = 0$ 
4:   repeat
5:      $k \leftarrow k + 1$ .
6:     Normalize  $v_j^{(k-1)}$ .
7:     (1) : Compute  $v_j^{(k)}$  by using  $v_j^{(k-1)}$ .
8:     if  $|\tilde{\lambda}_j - \tilde{\lambda}_{j-1}| \leq 10^{-3}\|T\|$ , then
9:       for  $i = j_1$  to  $j - 1$  do
10:         $v_j^{(k)} \leftarrow v_j^{(k)} - \langle v_j^{(k)}, v_i \rangle v_i$ 
11:      end for
12:    else
13:       $j_1 = j$ 
14:    end if
15:  until some condition is met.
16:  Normalize  $v_j^{(k)}$  to  $v_j$ .
17: end for

```

Fig. 1: Classical inverse iteration algorithm. j_1 means the index j of the first eigenvalue of cluster.

2.2 The defect of the classical inverse iteration algorithm

The inverse iteration is a prominent method for computing eigenvectors, because we can compute eigenvectors independently and this process is easily parallelized. When we use the classical inverse iteration on parallel computers, we can parallelize it even if some clusters exist.

Let us consider the Peters-Wilkinson method in the classical inverse iteration. When the dimension of T is greater than 1000, most of the eigenvalues are regarded as being in the same cluster [3].

In this case, we have to parallelize the inverse iteration with respect to not the cluster but the loop described from lines 1 to 17 in Figure 1. This loop includes the iteration based on Eq.(1) and the orthogonalization of the eigenvectors. This orthogonalization process becomes a bottleneck of the classical inverse iteration with respect to the computational time. The MGS algorithm is mainly based on a BLAS level-1 operation such as the inner product operation and the AXPY operation, and it is a sequential algorithm. Because of this, when we compute all the eigenvectors in parallel computers, the number of synchronizations is of order m^2 . Therefore, the MGS algorithm is ineffective on parallel computing.

In conclusion, the classical inverse iteration is an ineffective algorithm for parallel computing because the MGS algorithm is used in its orthogonalization process

3. Other orthogonalization algorithms

3.1 Householder orthogonalization

The Householder orthogonalization, based on the Householder matrices, is one of the alternative orthogonalization methods. When some vectors \mathbf{v} , $\mathbf{d} \in \mathbb{R}^n$ satisfy $\|\mathbf{v}\|_2 = \|\mathbf{d}\|_2$, there exists the symmetric matrix H satisfying $HH^\top = H^\top H = I$, $H\mathbf{v} = \mathbf{d}$ defined by

$$H = I - t\mathbf{y}\mathbf{y}^\top, \quad \mathbf{y} = \mathbf{v} - \mathbf{d}, \quad t = \frac{2}{\|\mathbf{y}\|_2^2}. \quad (2)$$

The transformation by H is called the Householder transformation. We can orthogonalize some vectors by using the Householder transformations. The algorithm of the Householder transformations is shown in Figure 2. The vector \mathbf{y}_j is the vector in which the elements from 1 to $(j-1)$ are the same as the elements of \mathbf{v}'_j and the elements from $(j+1)$ to n are zero. The vector \mathbf{e}_j is the j th vector of an n -dimensional identity matrix. In this paper, we call this algorithm the Householder orthogonalization.

```

1: for  $j = 1$  to  $m$  do
2:   Generate  $\mathbf{v}_j$  from  $\mathbf{q}_1, \dots, \mathbf{q}_{j-1}$ .
3:    $\mathbf{v}'_j = (I - t_{j-1}\mathbf{y}_{j-1}\mathbf{y}_{j-1}^\top) \cdots (I - t_2\mathbf{y}_2\mathbf{y}_2^\top) (I - t_1\mathbf{y}_1\mathbf{y}_1^\top) \mathbf{v}_j$ .
4:   Compute  $\mathbf{y}_j$  and  $t_j$  by using  $\mathbf{v}'_j$ .
5:    $\mathbf{q}_j = (I - t_1\mathbf{y}_1\mathbf{y}_1^\top) (I - t_2\mathbf{y}_2\mathbf{y}_2^\top) \cdots (I - t_j\mathbf{y}_j\mathbf{y}_j^\top) \mathbf{e}_j$ .
6: end for

```

Fig. 2: Householder orthogonalization algorithm.

The orthogonality of the vectors \mathbf{q}_j generated by the Householder orthogonalization does not depend on the condition number of the matrix T . Therefore, the Householder orthogonalization is more stable than the MGS. On the other hand, being similar to the MGS, it is a sequential algorithm that is mainly based on a BLAS level-1 operation. Its computational cost is higher than that of the MGS. Thus the Householder orthogonalization algorithm is an ineffective algorithm in parallel computing.

3.2 Compact WY orthogonalization

In 2010, Yamamoto presented the Householder orthogonalization in the Arnoldi process in terms of the compact WY representation [13]. This study suggests that the Householder orthogonalization becomes capable of computation with a BLAS level 2 operation in terms of the compact WY representation [11]. Yamamoto also showed that the computation time for orthogonalization on parallel computers has decreased with the use of the Householder orthogonalization algorithm in terms of the compact WY representation, compared to this computational time in the case of the MGS algorithm [13]. Although Yamamoto mainly shows the new

representation of orthogonalization in [13], in this paper, we show the implementation of this orthogonalization to the inverse iteration algorithm, and we evaluate its performance.

Now, we consider the Householder orthogonalization in Figure 2 and we introduce the compact WY representation. First, we define $Y_1 = \mathbf{y}_1 \in \mathbb{R}^{n \times j}$ and $T_1 = t_1 \in \mathbb{R}^{1 \times 1}$. Next, we define matrices $Y_j \in \mathbb{R}^{n \times j}$ and upper triangular matrices $T_j \in \mathbb{R}^{j \times j}$ recursively as follows:

$$Y_j = [Y_{j-1} \quad \mathbf{y}_j], T_j = \begin{bmatrix} T_{j-1} & -t_j T_{j-1} Y_{j-1}^\top \mathbf{y}_j \\ \mathbf{0} & t_j \end{bmatrix}. \quad (3)$$

In this case, the following equation holds

$$H_1 H_2 \cdots H_j = I - Y_j T_j Y_j^\top. \quad (4)$$

As shown by Eq.(4), we can rewrite the product of the Householder matrices $H_1 H_2 \cdots H_j$ in a simple block matrix form. Here $I - Y_j T_j Y_j^\top$ is called the compact WY representation of the product of the Householder matrices. Figure 3 shows the orthogonalization algorithm. Hereinafter, we refer to this orthogonalization algorithm as the compact WY orthogonalization.

```

1: for  $j = 1$  to  $m$  do
2:   Generate  $\mathbf{v}_j$  from  $\mathbf{q}_1, \dots, \mathbf{q}_{j-1}$ .
3:    $\mathbf{v}'_j = (I - Y_{j-1} T_{j-1}^\top Y_{j-1}^\top) \mathbf{v}_j$ .
4:   Compute  $\mathbf{y}_j$  and  $t_j$  by using  $\mathbf{v}'_j$ .
5:   (3) : Update  $Y_j$  and  $T_j$  by using  $t_j$ ,  $\mathbf{y}_j$ ,  $T_{j-1}$  and  $Y_{j-1}$ .
6:    $\mathbf{q}_j = (I - Y_j T_j Y_j^\top) \mathbf{e}_j$ .
7: end for

```

Fig. 3: Householder orthogonalization algorithm in terms of the compact WY representation.

3.3 Comparison of the orthogonalization algorithms

The compact WY orthogonalization has a stable orthogonality arising from the Householder transformations, and its mathematical calculation is mainly performed by BLAS level-2 operations 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. Table 1 displays the differences in performance of the three orthogonalization methods, considered in the above sections. In this table, ‘‘House’’ denotes the Householder orthogonalization and ‘‘cWY’’ denotes the compact WY orthogonalization. *Computation* denotes the order of the computational cost. *Synchronization* denotes the order of the number of synchronizations. *Orthogonality* denotes the norm $\|Q^\top Q - I\|$, where $Q = [\mathbf{q}_1, \dots, \mathbf{q}_n]$. ϵ denotes the machine epsilon and κ denotes the condition number of a matrix. These are the results obtained from [1] and [13].

Table 1: Comparison of the orthogonalization methods [1] [13].

methods	Computation	Synchronization	Orthogonality
MGS	$O(2m^2n)$	$O(m^2)$	$O(\epsilon\kappa(A))$
House	$O(4m^2n)$	$O(m^2)$	$O(\epsilon)$
cWY	$O(4m^2n)$	$O(m)$	$O(\epsilon)$

4. Inverse iteration method with the compact WY orthogonalization

In this section, we present a new inverse iteration algorithm. This new algorithm is described in Figure 4 and is based on DSTEIN, a LAPACK code of the classical inverse iteration. We change the orthogonalization process from the MGS to the Householder transformation in terms of the compact WY representation. In other words, we rewrite the MGS algorithm (from line 4 to 15 in Figure 1) to the compact WY orthogonalization algorithm shown in Figure 3.

Next, we explain an application of the compact WY orthogonalization to the classical inverse iteration. For the DSTEIN algorithm, we need not know the index j_c which denotes the j_c -th eigenvalue of the cluster in computing the j_c -th eigenvector. However, we must know the index for the compact WY orthogonalization when we compute and update T_j, Y_j . To overcome the above difficulty, we introduce a variable j_c on line 9, and we can recognize it.

This introduction of j_c enables us to execute the intended program. However, we do not get accurate results because the compact WY orthogonalization algorithm includes many equations with a comparatively large number of elements such as $Y_{j_c} T_{j_c}^T Y_{j_c}^T$ and $Y_{j_c} T_{j_c} Y_{j_c}^T$ and they may cause overflow. To overcome this difficulty, we have to normalize $v_j^{(k)}$ on line 6, and this normalization excludes overflow.

Finally, to reduce the computational cost, we transform parts of the equations. There are some examples in Figure 4 for $j_c = 1$, i.e., $j = 2$ in Figure 3.

In the original DSTEIN algorithm, we need not know that λ_{j_1} ($j_1 = j - 1$) is the first eigenvalue of the cluster. However, we must compute \mathbf{y}_1 and t_1 . Therefore, at the starting point of the computation of the eigenvector associated with the second eigenvalue, we compute \mathbf{y}_1 and t_1 . At this time, because T_1 is a 1×1 matrix, i.e., a scalar, we can omit the computation of some of Eq.(3) and only compute them. In addition, because we normalize $v_j^{(k-1)}$ on line 6 so that $v_j^{(k-1)} = \mathbf{q}_1$, we need not compute \mathbf{y}_1 again. As shown in lines 15 and 17, to save the computation step that is required when using BLAS, we change the formula from the matrix-vector operations to the vector operations. In addition, we implemented another formula because of the benefit of using BLAS computations to reduce the computational cost in line 23.

```

1: for  $j = 1$  to  $n$  do
2:   Generate  $v_j^{(0)}$  from random numbers.
3:    $k = 0$ 
4:   repeat
5:      $k \leftarrow k + 1$ .
6:     Normalize  $v_j^{(k-1)}$ .
7:     (1) : Compute  $v_j^{(k)}$  by using  $v_j^{(k-1)}$ .
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  $Y_1 = \mathbf{y}_1$  and  $T_1 = t_1$  by using  $v_{j_1-1}$ .
12:      end if
13:      Normalize  $v_j^{(k)}$ .
14:      if  $j_c = 1$ , then
15:         $v'_2 \leftarrow v_2 - t_1 \langle \mathbf{y}_1, v_j^{(k)} \rangle \mathbf{y}_1$ .
16:        Compute  $\mathbf{y}_2$  and update  $Y_2$  by using  $v'_2$ .
17:        Compute  $t_2$  and  $T_{1,2} = -t_2 t_1 \langle \mathbf{y}_1, \mathbf{y}_2 \rangle$  and update  $T_2$ .
18:      else
19:         $v'_{j_c+1} = (I - Y_{j_c} T_{j_c}^T Y_{j_c}^T) v_j^{(k)}$ .
20:        Compute  $\mathbf{y}_{j_c+1}$  and  $t_{j_c+1}$  by using  $v'_{j_c+1}$ .
21:        (3) : Update  $Y_{j_c+1}$  and  $T_{j_c+1}$  by using  $t_{j_c+1}, \mathbf{y}_{j_c+1},$ 
            $T_{j_c}$  and  $Y_{j_c}$ .
22:      end if
23:       $v_j^{(k)} \leftarrow (I - Y_{j_c+1} T_{j_c+1} Y_{j_c+1}^T) e_{j_c+1}$ .
24:    else
25:       $j_1 \leftarrow j$ .
26:    end if
27:  until some condition is met.
28:  Normalize  $v_j^{(k)}$  to  $v_j$ .
29: end for

```

Fig. 4: Algorithm of the compact WY inverse iteration.

5. Numerical experiments

In this section, we describe some numerical experiments performed using DSTEIN and DSTEIN-cWY in parallel computers, and we compare the computation time. DSTEIN is implemented in the classical inverse iteration algorithm, and DSTEIN-cWY is implemented in the new inverse iteration algorithm presented in the previous section.

5.1 Contents of the numerical experiments

In this subsection, we report computations of all the eigenvectors associated with eigenvalues of some matrices by using DSTEIN and DSTEIN-cWY on parallel computers, and we compare the calculation time. In these experiments, we compute the approximate eigenvalues by using LAPACK's program DSTEBZ, which is capable of computing them by using the bisection method. We record the calculation time for DSTEIN and DSTEIN-cWY using TIME, which is the internal function of Fortran and returns an integer number of times.

In the experiments, we use two computers equipped with multicore CPUs, and we implement those algorithms by using GotoBLAS2 [6], which is implemented to parallelize BLAS operations by assigning them to each CPU core. Table 2 shows the specifications of two computers.

All the matrices in the experiments are the glued-Wilkinson matrices W_j^\dagger , which are real symmetric and have

Table 3: DSTEIN and DSTEIN-cWY on Computer 1. Here, n is the dimension of the glued-Wilkinson matrix, t and t_{cwy} are computation time (sec.) by DSTEIN and DSTEIN-cWY on Computer 1, respectively.

n	1050	2010	3150	4200	5250	6300	7350	8400	9450	10500
t	2	9	25	55	106	178	276	400	560	758
t_{cwy}	1	2	5	10	16	25	37	57	81	113
t/t_{cwy}	2.00	4.50	5.00	5.50	6.63	7.12	7.46	7.02	6.91	6.71

Table 4: DSTEIN and DSTEIN-cWY on Computer 2. Here, n is the dimension of the glued-Wilkinson matrix, t and t_{cwy} are computation time (sec.) by DSTEIN and DSTEIN-cWY on Computer 2, respectively.

n	1050	2010	3150	4200	5250	6300	7350	8400	9450	10500
t	1	3	8	19	37	67	109	159	225	309
t_{cwy}	1	1	3	6	13	25	45	73	107	152
t/t_{cwy}	1.00	3.00	2.67	3.16	2.84	2.68	2.42	2.17	2.10	2.03

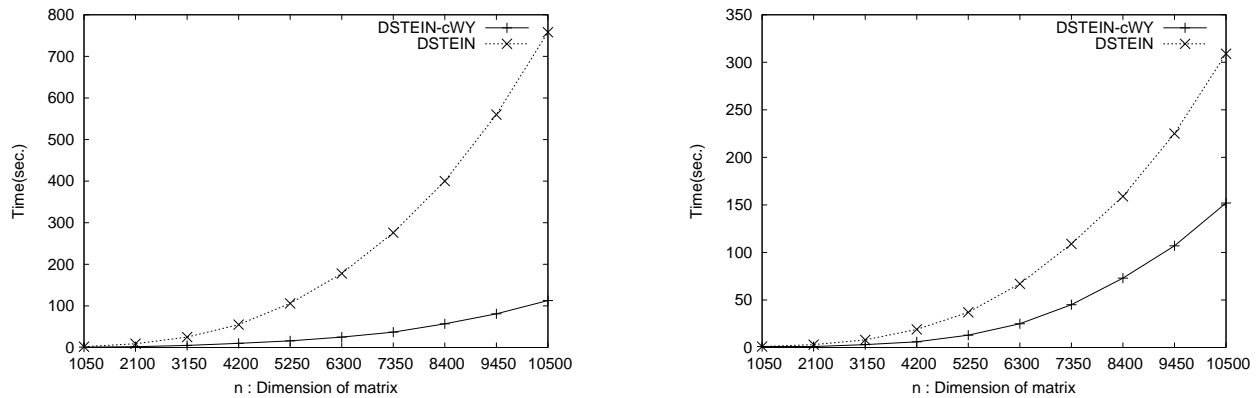


Fig. 5: Dimension n of the glued-Wilkinson matrix and the computation time by DSTEIN and DSTEIN-cWY. the above graph corresponds to Computer1 and the below Computer 2, respectively.

greater than that of the MGS orthogonalization where the classical inverse iteration is used. As the number of cores of the CPU increases, the parallelization efficiency increases.

In future studies, we will try to apply the new inverse iteration algorithms to other types of matrix eigenvector problem, such as eigenvectors of a real symmetric banded matrix, or singular vectors of a bidiagonal matrix.

Acknowledgements.

The authors thank Professor Yusaku Yamamoto of Kobe University for providing several helpful suggestions.

References

- [1] J. W. Demmel, L. Grigori, M. Hoemmen and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, LAPACK Working Notes, No.204, 2008.
- [2] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vömel, *Performance and accuracy of LAPACK's symmetric tridiagonal eigensolvers*, SIAM J. Sci. Comput., Vol. 30, No. 3, pp. 1508-1526, 2008.
- [3] I. S. 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.
- [4] I. S. Dhillon, B. N. Parlett, and C. Vömel, *Glued matrices and the MRRR algorithm*, SIAM J. Sci. Comput., Vol. 27, No. 2, pp. 496-510, 2005.
- [5] G. Golub and C. van Loan, *Matrix Computations*, Johns Hopkins Univ. Press, 1996.
- [6] GotoBLAS2, <http://www.tacc.utexas.edu/tacc-projects/gotoblas2/>.
- [7] 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.
- [8] M. Gu and S. C. Eisenstat, *A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem*, SIAM J. Mat. Anal. Appl., Vol. 16, pp. 172-191, 1995.
- [9] LAPACK, <http://www.netlib.org/lapack/>.
- [10] 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.
- [11] 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.
- [12] H. Walker, *Implementation of the GMRES method using Householder transformations*, SIAM J. Sci. Stat. Comput., Vol. 9, No. 1, pp. 152-163, 1988.
- [13] Y. Yamamoto, *Parallelization of orthogonalization in Arnoldi process based on the compact WY representation*, Proceedings of the Annual Conference of JSIAM, pp. 39-40, 2010 (in Japanese).

Resultant-factorization Technique for Obtaining Solutions to Ordinary Differential Equations

Kinji Kimura* and Hiroshi Yoshida†

* Graduate School of Informatics, Kyoto University,
36-1 Yoshida-Honmachi, Sakyo-ku, Kyoto 606-8501, Japan.

† Faculty of Mathematics, Kyushu University,
Ito, Motouoka 744, Nishi-ku, Fukuoka 819-0395, Japan.

Abstract—We propose a technique for obtaining solutions to ordinary differential equations. A system of differential equations sometimes has multiple solutions with distinct features. Prime ideal decomposition can be used for extracting the desired solution from these solutions. Solutions to algebraic equations contain many parameters, and in such a case, prime ideal decomposition is less tractable. As an alternative, we propose a resultant-factorization technique for extracting the desired solution. We also demonstrate the implementation of this technique and show its timing data.

Keywords: resultant, factorization, prime ideal decomposition, Gröbner basis, saturation

1. Introduction

In ordinary differential equations, we often need to analyze complicated polynomials with multiple irreducible affine varieties (solutions). It is therefore essential to perform prime ideal decomposition. To understand the importance of the decomposition, consider the following set of polynomials: $\{x^3 - x^2y - xy - 2x + y^2 + 2y, xy - x - y^2 + y\}$. Prime ideal decomposition shows that the above polynomials can be decomposed into two irreducible affine varieties, $\{x - y\}$ and $\{x^2 - 3, y - 1\}$. While one cannot determine concrete values of x and y using the former, one can determine them, $x = \pm\sqrt{3}$ and $y = 1$, using the latter. Hence, to decide whether or not we can determine the values of variables, we must perform prime ideal decomposition of polynomials. In ordinary differential equations, however, there are many parameters and variables that describe observed data and systems, respectively. In general, large numbers of parameters and variables render prime ideal decomposition more difficult [4]. Especially, the method in [4] needs the computation of Gröbner basis in the beginning. Under large numbers of parameters and variables render, the computational cost of Gröbner basis increases. This leads us to propose a *resultant-factorization* technique [6] that provides the desired irreducible affine variety (solution). By using the resultant-factorization technique, we can reduce the computational cost of Gröbner basis.

2. Resultant-factorization technique

On the basis of [1], we propose an efficient technique to arrive at the targeted affine variety (solution); the technique is shown in Fig. 1(a).

Let $BP = \{BP_i \mid 1 \leq i \leq n\}$ be an original set of polynomials. Let F_1 be a set of $n - 1$ resultants of polynomials BP_j ($1 \leq j \leq n, i \neq j$) for some BP_i ($1 \leq i \leq n$) in a variable, say, x_1 . It is reasonable to select a variable x_1 such that BP_i and BP_j are low-degree polynomials in x_1 . For instance, if one can factorize some element f in F_1 into mutually disjoint elements f_1, f_2 , and f_3 , $\langle F_1 \rangle$ can be decomposed into $\langle F_1, f_1 \rangle \cap \langle F_1, f_2 \rangle \cap \langle F_1, f_3 \rangle$. As a result of the factorization, the resultant F_{21} in x_2 can usually be described by a smaller set of polynomials. Furthermore, when one obtains a factorized term like $(y_1 + y_2^2)(z - y_3^2 + 3)^3$ with the rate constants y_1 and y_2 , it is sufficient to consider only $(z - y_3^2 + 3)$ because the rate constants are guaranteed to be positive ($y_1 > 0 \wedge y_2 > 0 \Rightarrow y_1 + y_2^2 > 0$) and the radical $\sqrt{\langle BP \rangle}$ suffices. We can also ignore a factor like $(y_1 - y_2)$ when we assume $y_1 \neq y_2$ because of unacceptable factors or mathematically trival factors. These simplifications allow us to prune the branches of the resultant-factorization series¹. When we arrive at an appropriate ideal ($\langle F_{32} \rangle$ in Fig. 1(a)), we can efficiently determine all the rate constants by using the ideal $\langle BP \rangle + \langle F_{32} \rangle$, as illustrated in Fig. 1(a).

3. Implementation

We have implemented our “Resultant-factorization technique,” which is shown as “ScodeRFP.rr” on <http://sites.google.com/site/codes86/>. The main routine of “ScodeRFP.rr” is “automatic_decom,” which calls the following seven procedures:

- 1) Procedure 1: `shortcut_speed(ideal I, list of polynomials lp)` returns an ideal obtained by removing the factors in lp from ideal I . This procedure aims to remove unacceptable factors or mathematically trival factors.
- 2) Procedure 2: `idealclean(ideal I)` returns an ideal obtained by removing redundant elements from ideal I .

¹Recently, such a pruning procedure in algebraic approaches has been studied, e.g., “positive quantifier elimination”(QE) [5].

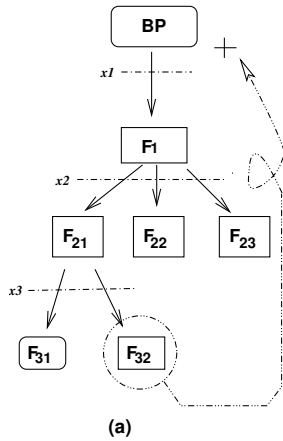


Figure 1: Schematic illustration of the resultant-factorization technique

- 3) Procedure 3: coefficientcleaner(ideal I , poly p) returns an ideal obtained by removing common factors and p from ideal I .
- 4) Procedure 4: constant_check(ideal I) checks whether or not ideal I has an element composed of only the parameters. If there is such an element, the function returns 0 to indicate the presence of an error in the top-level function.
- 5) Procedure 5: idealfactorize returns a list of elements that can be factorized over \mathbb{Q} in a given ideal; if there are no such elements in the ideal, then it returns the given ideal. This is based on the relation $\sqrt{\langle I, f * g \rangle} = \sqrt{\langle I, f \rangle} \cap \sqrt{\langle I, g \rangle}$.
- 6) Procedure 6: variable_choice returns a variable that is to be removed in the next procedure. There are two types of outputs. Let lp be a given list of polynomials.
 - a) In lp , when there is a variable contained in only one polynomial, “variable_choice” returns the variable. In this case, it is not necessary to calculate resultants in the next procedure because the resultant of polynomials p and q in x is q^r , where q does not contain the variable x and r is the degree of x in p .
 - b) Otherwise, we select a variable as follows:
 - i) We calculate d_i – the maximum degree of variable $x_i (1 \leq i \leq n)$ by considering lp . If a single d_j is the minimum among $d_i (1 \leq i \leq n)$, x_j is returned.
 - ii) If multiple d_i 's have the same minimum value, let y_1, y_2, \dots, y_m be variables that provide this minimum. We calculate n_i , the number of polynomials that contain y_i . If a single n_j provides the minimum among $n_i (1 \leq i \leq m)$, y_j is returned.
 - iii) If multiple n_i 's provide the same minimum, let z_1, z_2, \dots, z_k be the variables that provide

this minimum. We calculate t_i which means the number of terms in the polynomials that contain z_i . Variable z_j that provides the minimum and that is calculated first is returned.

In (i)-(iii), as an accompanying output, we return the polynomial that contains the returned variable and has the minimum number of terms.

- 7) Procedure 7: idealresultant returns a set of resultants on the basis of the output of Procedure 6.

We perform Procedures 1,2,...,7 for a given input ideal. Procedure 5 gives rise to branches of Procedures where the main routine is recursively called. Finally, the main routine returns a list of polynomials together with the input original ideal. Prime ideal decomposition can be rapidly performed for each of the polynomials in the list.

4. Model

In this section, we introduce the Painlevé VI equation [2]

$$\begin{aligned}
 y'' &= \frac{1}{2} \left(\frac{1}{y} + \frac{1}{y-1} + \frac{1}{y-t} \right) y'^2 \\
 &- \left(\frac{1}{t} + \frac{1}{t-1} + \frac{1}{y-t} \right) y' \\
 &+ \frac{y(y-1)(y-t)}{t^2(t-1)^2} \times \\
 &\left(\frac{\alpha_1^2}{2} - \frac{\alpha_4^2 t}{2 y^2} + \frac{\alpha_3^2 (t-1)}{2 (y-1)^2} \right. \\
 &\left. - \frac{\alpha_0^2 - 1 t(t-1)}{2 (y-t)^2} \right), \tag{1}
 \end{aligned}$$

which is an ordinary differential equation. Equation (1) has four parameters: $\alpha_0, \alpha_1, \alpha_3,$ and α_4 . We discuss typical solutions, which are called “rational solutions.” All rational solutions for the equation have been obtained in [3]. We can use algebraic geometry for studying the equation (1). However, in general, we cannot use these techniques for all ordinary differential equations. Thus, we treat equation (1) in the condition that we do not know the mathematical structure of the equation (1). We assume rational solutions of the form $y(t) = (k_0 + k_1 t + k_2 t^2)/(l_0 + t)$ and substitute these solutions in the equation. We then obtain algebraic equations for the eight variables $k_0, k_1, k_2, l_0, \alpha_0, \alpha_1, \alpha_3,$ and α_4 . By using the resultant-factorization technique proposed in the previous section 2, we can determine the solutions.

We assume rational solutions of the form $y(t) = (k_0 + k_1 t + k_2 t^2)/(l_0 + t)$ and substitute these solutions in the

equation (1). Then, we get the following ideal,

$$\begin{aligned}
 P = & \{-k_0^4 \times \\
 & (\alpha_1 k_0 + (-\alpha_1 - \alpha_3)l_0) \times \\
 & (\alpha_1 k_0 + (-\alpha_1 + \alpha_3)l_0), \\
 & \dots, \\
 & -k_2^4 \times \\
 & (\alpha_1 k_2 - \alpha_1 + \alpha_0) \times \\
 & (\alpha_1 k_2 - \alpha_1 - \alpha_0)\} \\
 = & \{p_1, p_2, \dots, p_{13}\} = 0. \tag{2}
 \end{aligned}$$

For convenience, we define five equations: $f_1 = \alpha_1 k_0 + (-\alpha_1 + \alpha_3)l_0$, $f_2 = \alpha_1 k_0 + (-\alpha_1 - \alpha_3)l_0$, $g_1 = \alpha_1 k_2 - \alpha_1 + \alpha_0$, $g_2 = \alpha_1 k_2 - \alpha_1 - \alpha_0$, and $h = k_0 - l_0 k_1 + l_0^2 k_2$. Hence, we can divide the original problem($P = 0$) to the six cases,

$$\begin{aligned}
 P_1 = & \{f_1, p_2, \dots, p_{12}, g_1\} = 0, \\
 P_2 = & \{f_1, p_2, \dots, p_{12}, g_2\} = 0, \\
 P_3 = & \{f_2, p_2, \dots, p_{12}, g_1\} = 0, \\
 P_4 = & \{f_2, p_2, \dots, p_{12}, g_2\} = 0, \\
 P_5 = & \{k_0, p_2, \dots, p_{12}, p_{13}\} = 0, \\
 P_6 = & \{p_1, p_2, \dots, p_{12}, k_2\} = 0.
 \end{aligned}$$

We explain the resultant-factorization technique for the case of $P_1 = 0$ in detail. Thus, we consider the ideal,

$$\langle f_1, p_2, \dots, p_{12}, g_1 \rangle.$$

We compute solutions in the case of $f_1 = 0$ and $g_1 = 0$ as follows,

1) Step 1: we compute,

$$\begin{aligned}
 Q_1^{(0)} = & \text{resultant}_{\alpha_0}(g_1, f_1), \\
 Q_2^{(0)} = & \text{resultant}_{\alpha_0}(g_1, p_2), \\
 & \dots \\
 Q_{12}^{(0)} = & \text{resultant}_{\alpha_0}(g_1, p_{12}).
 \end{aligned}$$

We factorize $Q_{12}^{(0)}$ and set $Q_1^{(1)}, \dots, Q_{12}^{(1)}$,

$$\begin{aligned}
 Q_1^{(1)} = & Q_1^{(0)}, \\
 & \dots \\
 Q_{11}^{(1)} = & Q_{11}^{(0)}, \\
 Q_{12}^{(1)} = & (k_2 - 1)Q_{12}^{(1)}, \tag{3}
 \end{aligned}$$

$$Q_{12}^{(1)} = \frac{Q_{12}^{(0)}}{k_2 - 1}.$$

2) Step 2: we compute,

$$\begin{aligned}
 R_2^{(0)} = & \text{resultant}_{\alpha_3}(Q_1^{(1)}, Q_2^{(1)}), \\
 & \dots \\
 R_{12}^{(0)} = & \text{resultant}_{\alpha_3}(Q_1^{(1)}, Q_{12}^{(1)}).
 \end{aligned}$$

We factorize $R_2^{(0)}$ and set $R_2^{(1)}, \dots, R_{12}^{(1)}$,

$$\begin{aligned}
 R_2^{(0)} = & l_0(k_0 - l_0)R_2^{(1)}, \tag{4} \\
 R_2^{(1)} = & \frac{R_2^{(0)}}{l_0(k_0 - l_0)} \\
 R_3^{(1)} = & R_3^{(0)}, \\
 & \dots \\
 R_{12}^{(1)} = & R_{12}^{(0)}.
 \end{aligned}$$

3) Step 3: we compute,

$$\begin{aligned}
 S_2^{(0)} = & \text{resultant}_{\alpha_4}(R_{12}^{(1)}, R_2^{(1)}), \\
 & \dots \\
 S_{11}^{(0)} = & \text{resultant}_{\alpha_4}(R_{12}^{(1)}, R_{11}^{(1)}).
 \end{aligned}$$

We factorize $S_2^{(0)}, \dots, S_{11}^{(0)}$ and set $S_2^{(1)}, \dots, S_{11}^{(1)}$,

$$\begin{aligned}
 S_2^{(0)} = & 4l_0^4 \left(S_2^{(1)}\right)^2, S_2^{(1)} = \sqrt{\frac{S_2^{(0)}}{4l_0^4}} \\
 S_3^{(0)} = & l_0^4 \left(S_3^{(1)}\right)^2, S_3^{(1)} = \sqrt{\frac{S_3^{(0)}}{l_0^4}} \\
 & \dots \\
 S_{10}^{(0)} = & l_0^4 \left(S_{10}^{(1)}\right)^2, S_{10}^{(1)} = \sqrt{\frac{S_{10}^{(0)}}{l_0^4}} \\
 S_{11}^{(0)} = & l_0^4 (k_2 - 1)^2 \left(S_{11}^{(1)}\right)^2, \\
 S_2^{(1)} = & \sqrt{\frac{S_{11}^{(0)}}{l_0^4 (k_2 - 1)^2}}
 \end{aligned}$$

4) Step 4: we compute,

$$\begin{aligned}
 T_3^{(0)} = & \text{resultant}_{\alpha_1}(S_2^{(1)}, S_3^{(1)}), \\
 & \dots \\
 T_{11}^{(0)} = & \text{resultant}_{\alpha_1}(S_2^{(1)}, S_{11}^{(1)}).
 \end{aligned}$$

We factorize $T_2^{(0)}, \dots, T_{11}^{(0)}$ and set $T_3^{(1)}, \dots, T_{11}^{(1)}$,

$$\begin{aligned}
 T_3^{(0)} = & l_0^6 (k_2 - 1)^2 (k_0 - l_0)^2 h^2 \left(T_3^{(1)}\right)^2, \\
 T_3^{(1)} = & \sqrt{\frac{T_3^{(0)}}{l_0^6 (k_2 - 1)^2 (k_0 - l_0)^2 h^2}} \\
 T_4^{(0)} = & l_0^4 (k_2 - 1)^2 h^2 \left(T_4^{(1)}\right)^2, \\
 T_4^{(1)} = & \sqrt{\frac{T_4^{(0)}}{l_0^4 (k_2 - 1)^2 h^2}} \\
 & \dots \\
 T_{11}^{(0)} = & l_0^4 (k_2 - 1)^2 h^2 \left(T_{11}^{(1)}\right)^2, \\
 T_{11}^{(1)} = & \sqrt{\frac{T_{11}^{(0)}}{l_0^4 (k_2 - 1)^2 h^2}}.
 \end{aligned}$$

- 5) Step 5: from the process of the above-mentioned computation, we can assume the following conditions,

$$k_2 \neq 1, l_0 \neq 0, k_0 \neq l_0, h \neq 0, k_0 \neq 0.$$

- 6) Step 6: we compute the Gröbner basis(G_0) [4] of the ideal,

$$\langle T_3^{(1)}, \dots, T_{11}^{(1)} \rangle.$$

- 7) Step 7: by using saturation technique [4], we remove the component $k_2 - 1$ from the ideal(G_0),

$$G_1 = \langle G_0, 1 - u(k_2 - 1) \rangle.$$

- 8) Step 8: by using saturation technique, we remove the component l_0 from the ideal(G_1),

$$G_2 = \langle G_1, 1 - u(l_0) \rangle.$$

- 9) Step 9: by using saturation technique, we remove the component $k_0 - l_0$ from the ideal(G_2),

$$G_3 = \langle G_2, 1 - u(k_0 - l_0) \rangle.$$

- 10) Step 10: by using saturation technique, we remove the component h from the ideal(G_3),

$$G_4 = \langle G_3, 1 - u(h) \rangle.$$

- 11) Step 11: by using saturation technique, we remove the component k_0 from the ideal(G_3),

$$G_5 = \langle G_4, 1 - u(k_0) \rangle.$$

- 12) Step 12: we can get the ideal(G_6) by using computation of the following ideal,

$$G_6 = \langle G_5, P_1 \rangle.$$

- 13) Step 13: we get solutions(D_1) from G_6 by using prime ideal decomposition.

5. Timing data

If we do not use the resultant-factorization technique, we cannot obtain all the solutions for the ideal (2) within 48 h. However, we can compute solutions with Table 1 by using the resultant-factorization technique. Since the solutions of the equation (1) has a large number of irreducible affine varieties, the resultant-factorization technique functions effectively. Table 1 shows the computing times for the different cases. All the algorithms are implemented on Risa/Asir² and measurements are performed on a PC with Intel Core i7 920 and 12 GB of main memory.

²<http://www.math.kobe-u.ac.jp/Asir/asir.html>

Table 1: computing time for different cases

Cases	Time	Solutions
$f_1 = 0 \ \& \ g_1 = 0$	4m34s	D_1
$f_1 = 0 \ \& \ g_2 = 0$	4m46s	D_2
$f_2 = 0 \ \& \ g_1 = 0$	4m35s	D_3
$f_2 = 0 \ \& \ g_2 = 0$	4m41s	D_4

Cases	Time	Solutions
$k_0 = 0$	351m56s	D_5
$l_0 = 0$	45s	D_6
$k_2 = 0$	146m14s	D_7

Cases	Time	Solutions
$k_2 = 1$	7s	D_8
$k_0 = l_0$	78m57s	D_9
$h = 0$	3s	D_{10}

6. Solutions

We show $D_1, D_5, D_6, D_7, D_8, D_9$ and D_{10} .

$$D_1 = \{ \{ \alpha_1 + 2, k_1 + 2k_2 + \alpha_4, -k_1 + \alpha_3 + 1, 2k_2 - \alpha_0 - 2, -k_1 + 2l_0k_2 - l_0, -2k_0 + l_0k_1 + l_0, (-4k_2 + 2)k_0 + k_1^2 + k_1 \}, \dots \\ \{ k_1 + 2k_2 - 2, k_0 - k_2 - l_0 + 1, \alpha_4 + 2, \alpha_1 - \alpha_3 - \alpha_0 + 1, \alpha_3l_0 - \alpha_1 + \alpha_3 - 1, -\alpha_1k_2 + \alpha_3 - 1 \}, \dots \} = 0$$

$$D_5 = \{ \{ l_0, k_2, -\alpha_3 + \alpha_4 + \alpha_1, \alpha_0 + 1, -\alpha_4 - k_1\alpha_1 \}, \dots \\ \{ l_0, k_1 + k_2 - 1, \alpha_4 + 1, \alpha_0 + \alpha_3 + \alpha_1, \alpha_3 + k_2\alpha_1 \}, \dots \\ \{ l_0, \alpha_1 + 1, \alpha_4 + k_1 + k_2, \alpha_3 + k_1 - 1, \alpha_0 + k_2 - 1 \}, \dots \\ \{ l_0, k_2, k_1 - 1, \alpha_3 \}, \\ \{ k_2, k_1 - l_0 - 1, \alpha_1 + 1, -\alpha_0 + \alpha_3 + \alpha_4, (l_0 + 1)\alpha_3 + l_0\alpha_4 \}, \dots \\ \{ k_2, \alpha_4 + 1, \alpha_0 + \alpha_3 + \alpha_1 + 2, (l_0 + 1)\alpha_3 + \alpha_1 + l_0 + 1, k_1\alpha_1 + l_0 + 1, k_1\alpha_3 + k_1 - 1 \}, \dots \\ \{ k_2, k_1 - l_0 - 1, \alpha_1 - 1, -\alpha_0 + \alpha_3 + \alpha_4, (l_0 + 1)\alpha_3 + l_0\alpha_4 \}, \dots \\ \{ k_2, \alpha_4 + 1, -\alpha_0 - \alpha_3 + \alpha_1 - 2, (l_0 + 1)\alpha_3 - \alpha_1 + l_0 + 1, k_1\alpha_1 - l_0 - 1, k_1\alpha_3 + k_1 - 1 \}, \dots \\ \{ k_1 - l_0k_2, \alpha_3 + 1, \alpha_0 - \alpha_4 + \alpha_1, -\alpha_4 + k_2\alpha_1 \}, \dots \\ \{ k_1 + k_2 - l_0 - 1, -\alpha_3 + \alpha_4 - \alpha_1 + 2, \alpha_0 + 1, \dots \} = 0$$

$$\begin{aligned}
 & -\alpha_4 - l_0\alpha_1 - 1, (k_2 - 1)\alpha_1 + 1, \\
 & (-k_2 + 1)\alpha_4 - k_2 + l_0 + 1\}, \dots \\
 & \{k_1 + 1, \alpha_3 + 2, \alpha_0 - \alpha_4 + \alpha_1 + 1, \\
 & l_0\alpha_4 + \alpha_1 + 1, -\alpha_4 + k_2\alpha_1 + 1\}, \dots \\
 & \{k_1 - l_0k_2, \alpha_3 + 1, -\alpha_0 + \alpha_4 + \alpha_1, \\
 & -\alpha_4 - k_2\alpha_1\}, \dots \\
 & \{k_1 + k_2 - l_0 - 1, -\alpha_3 - \alpha_4 - \alpha_1 + 2, \alpha_0 - 1, \\
 & \alpha_4 - l_0\alpha_1 - 1, (k_2 - 1)\alpha_1 + 1, \\
 & (k_2 - 1)\alpha_4 - k_2 + l_0 + 1\}, \dots \\
 & \{k_1 + 1, \alpha_3 + 2, -\alpha_0 + \alpha_4 + \alpha_1 + 1, \\
 & -l_0\alpha_4 + \alpha_1 + 1, -\alpha_4 - k_2\alpha_1 - 1\}, \dots \\
 & \{k_2, k_1 - l_0 - 1, \alpha_1 + 1, \alpha_0 + \alpha_3 + \alpha_4, \\
 & (l_0 + 1)\alpha_3 + l_0\alpha_4\}, \dots \\
 & \{k_2, \alpha_4 + 1, -\alpha_0 + \alpha_3 + \alpha_1 + 2, \\
 & (l_0 + 1)\alpha_3 + \alpha_1 + l_0 + 1, k_1\alpha_1 + l_0 + 1, \\
 & k_1\alpha_3 + k_1 - 1\}, \dots \\
 & \{k_2, k_1 - l_0 - 1, \alpha_1 - 1, \alpha_0 + \alpha_3 + \alpha_4, \\
 & (l_0 + 1)\alpha_3 + l_0\alpha_4\}, \dots \\
 & \{k_2, \alpha_4 + 1, \alpha_0 - \alpha_3 + \alpha_1 - 2, \\
 & (l_0 + 1)\alpha_3 - \alpha_1 + l_0 + 1, k_1\alpha_1 - l_0 - 1, \\
 & k_1\alpha_3 + k_1 - 1\}, \dots \\
 & \{k_2, k_1, \alpha_4\}, \\
 & \{k_1 - l_0k_2, \alpha_3 + 1, \alpha_0 + \alpha_4 + \alpha_1, \\
 & \alpha_4 + k_2\alpha_1\}, \dots \\
 & \{k_1 + k_2 - l_0 - 1, -\alpha_3 - \alpha_4 - \alpha_1 + 2, \\
 & \alpha_0 + 1, \alpha_4 - l_0\alpha_1 - 1, (k_2 - 1)\alpha_1 + 1, \\
 & (k_2 - 1)\alpha_4 - k_2 + l_0 + 1\}, \dots \\
 & \{k_1 + 1, \alpha_3 + 2, \alpha_0 + \alpha_4 + \alpha_1 + 1, \\
 & -l_0\alpha_4 + \alpha_1 + 1, \alpha_4 + k_2\alpha_1 + 1\}, \dots \\
 & \{k_1 - l_0k_2, \alpha_3 + 1, -\alpha_0 - \alpha_4 + \alpha_1, \\
 & \alpha_4 - k_2\alpha_1\}, \dots \\
 & \{k_1 + k_2 - l_0 - 1, -\alpha_3 + \alpha_4 - \alpha_1 + 2, \\
 & \alpha_0 - 1, -\alpha_4 - l_0\alpha_1 - 1, (k_2 - 1)\alpha_1 + 1, \\
 & (-k_2 + 1)\alpha_4 - k_2 + l_0 + 1\}, \dots \\
 & \{k_2 - 1, k_1 - l_0, \alpha_0\}, \\
 & \{k_1 + 1, \alpha_3 + 2, -\alpha_0 - \alpha_4 + \alpha_1 + 1, \\
 & l_0\alpha_4 + \alpha_1 + 1, \alpha_4 - k_2\alpha_1 - 1\}, \dots \} = 0
 \end{aligned}$$

$$\begin{aligned}
 D_6 = & \{\{k_2, k_0 + k_1 - 1, \alpha_1, \alpha_3 + 1, \\
 & \alpha_4 + \alpha_0 + 2, (\alpha_4 + 1)k_1 - \alpha_4\}, \dots \\
 & \{k_1 + 2k_2 - 2, k_0 - k_2 + 1, \alpha_1, \\
 & \alpha_4 + 2, \alpha_3 + 1, \alpha_0\}, \dots \\
 & \{k_1 + k_2 - 1, k_0, \alpha_4 + 1, \\
 & \alpha_1 + \alpha_3 + \alpha_0, \alpha_1k_2 + \alpha_3\}, \dots \\
 & \{k_2, k_0, \alpha_1 + \alpha_4 - \alpha_3, \alpha_0 + 1, -\alpha_1k_1 - \alpha_4\}, \dots \\
 & \{k_0, \alpha_1 + 1, k_1 + k_2 + \alpha_4, \\
 & k_1 - \alpha_3 - 1, k_2 + \alpha_0 - 1\}, \dots \\
 & \{k_1 + k_2 - 1, k_0, \alpha_4 + 1, \\
 & \alpha_1 + \alpha_3 - \alpha_0, -\alpha_1k_2 - \alpha_3\}, \dots \\
 & \{k_1, k_0, \alpha_3 + 1, \alpha_1 + \alpha_4 + \alpha_0, \alpha_1k_2 + \alpha_4\}, \dots \\
 & \{k_1, k_0, \alpha_3 + 1, \alpha_1 + \alpha_4 - \alpha_0, -\alpha_1k_2 - \alpha_4\}, \dots \\
 & \{k_2, k_1, \alpha_1, \alpha_4, \alpha_3 + 1, \alpha_0 + 2\}, \dots \\
 & \{k_2, k_1 - 1, k_0, \alpha_3\}, \{k_2, k_1, k_0, \alpha_4\}, \\
 & \{k_2 - 1, k_1, k_0, \alpha_0\}\} = 0
 \end{aligned}$$

$$\begin{aligned}
 D_7 = & \{\{k_0, \alpha_4 + 1, \alpha_1 + \alpha_3 + \alpha_0 + 2, \\
 & (\alpha_3 + 1)l_0 + \alpha_1 + \alpha_3 + 1, \alpha_1k_1 + l_0 + 1, \\
 & (\alpha_3 + 1)k_1 - 1\}, \dots \\
 & \{k_1 - l_0 - 1, k_0, \alpha_1 + 1, \alpha_4 - \alpha_3 + \alpha_0, \\
 & (\alpha_4 - \alpha_3)l_0 - \alpha_3\}, \dots \\
 & \{k_0, \alpha_4 + 1, \alpha_1 - \alpha_3 + \alpha_0 + 2, \\
 & (\alpha_3 - 1)l_0 - \alpha_1 + \alpha_3 - 1, \alpha_1k_1 + l_0 + 1, \\
 & (\alpha_3 - 1)k_1 + 1\}, \dots \\
 & \{k_1 - l_0 - 1, k_0, \alpha_1 + 1, \alpha_4 + \alpha_3 + \alpha_0, \\
 & (\alpha_4 + \alpha_3)l_0 + \alpha_3\}, \dots \\
 & \{k_1, k_0, \alpha_4\}, \\
 & \{k_0 - l_0k_1, \alpha_1 - \alpha_4 + \alpha_3, \\
 & \alpha_0 + 1, \alpha_1k_1 - \alpha_4\}, \dots \\
 & \{k_0 + k_1 - l_0 - 1, \alpha_3 - 1, \alpha_1 - \alpha_4 + \alpha_0 + 2, \\
 & (\alpha_4 - 1)l_0 + \alpha_1, \alpha_1k_1 + l_0 - \alpha_1, \\
 & (\alpha_4 - 1)k_1 - \alpha_4\}, \dots \\
 & \{k_1 + l_0, -\alpha_1 + \alpha_4 - \alpha_3 - 1, \alpha_0 + 2, \\
 & (-\alpha_1 - 1)l_0 - \alpha_4, -\alpha_1k_0 + (\alpha_4 - 1)l_0, \\
 & (-l_0 - \alpha_4)k_0 + (-\alpha_4 + 1)l_0^2\}, \dots \\
 & \{k_1 + l_0, -\alpha_1 + \alpha_4 - \alpha_3 + 1, \alpha_0 + 2, \\
 & (-\alpha_1 + 1)l_0 - \alpha_4, -\alpha_1k_0 + (\alpha_4 + 1)l_0, \\
 & (l_0 - \alpha_4)k_0 + (-\alpha_4 - 1)l_0^2\}, \dots \\
 & \{k_0 - l_0k_1, \alpha_1 + \alpha_4 + \alpha_3, \alpha_0 + 1, \\
 & \alpha_1k_1 + \alpha_4\}, \dots \\
 & \{k_0 + k_1 - l_0 - 1, \alpha_3 - 1, \alpha_1 + \alpha_4 + \alpha_0 + 2,
 \end{aligned}$$

$$\begin{aligned}
 &(\alpha_4 + 1)l_0 - \alpha_1, \alpha_1 k_1 + l_0 - \alpha_1, \\
 &(\alpha_4 + 1)k_1 - \alpha_4\}, \dots \\
 &\{k_1 + l_0, -\alpha_1 - \alpha_4 - \alpha_3 - 1, \alpha_0 + 2, \\
 &(-\alpha_1 - 1)l_0 + \alpha_4, -\alpha_1 k_0 + (-\alpha_4 - 1)l_0, \\
 &(-l_0 + \alpha_4)k_0 + (\alpha_4 + 1)l_0^2\}, \dots \\
 &\{k_1 + l_0, -\alpha_1 - \alpha_4 - \alpha_3 + 1, \alpha_0 + 2, \\
 &(-\alpha_1 + 1)l_0 + \alpha_4, -\alpha_1 k_0 + (-\alpha_4 + 1)l_0, \\
 &(l_0 + \alpha_4)k_0 + (\alpha_4 - 1)l_0^2\}, \dots \\
 &\{k_0 - l_0 k_1, \alpha_1 + \alpha_4 - \alpha_3, \alpha_0 + 1, \\
 &-\alpha_1 k_1 - \alpha_4\}, \dots \\
 &\{k_0 + k_1 - l_0 - 1, \alpha_3 + 1, \alpha_1 + \alpha_4 + \alpha_0 + 2, \\
 &(\alpha_4 + 1)l_0 - \alpha_1, \alpha_1 k_1 + l_0 - \alpha_1, \\
 &(\alpha_4 + 1)k_1 - \alpha_4\}, \dots \\
 &\{k_1 + l_0, -\alpha_1 - \alpha_4 + \alpha_3 - 1, \alpha_0 + 2, \\
 &(\alpha_1 + 1)l_0 - \alpha_4, \alpha_1 k_0 + (\alpha_4 + 1)l_0, \\
 &(-l_0 + \alpha_4)k_0 + (\alpha_4 + 1)l_0^2\}, \dots \\
 &\{k_1 + l_0, -\alpha_1 - \alpha_4 + \alpha_3 + 1, \alpha_0 + 2, \\
 &(\alpha_1 - 1)l_0 - \alpha_4, \alpha_1 k_0 + (\alpha_4 - 1)l_0, \\
 &(l_0 + \alpha_4)k_0 + (\alpha_4 - 1)l_0^2\}, \dots \\
 &\{k_0 - l_0 k_1, \alpha_1 - \alpha_4 - \alpha_3, \alpha_0 + 1, \\
 &-\alpha_1 k_1 + \alpha_4\}, \dots \\
 &\{k_0 + k_1 - l_0 - 1, \alpha_3 + 1, \alpha_1 - \alpha_4 + \alpha_0 + 2, \\
 &(\alpha_4 - 1)l_0 + \alpha_1, \alpha_1 k_1 + l_0 - \alpha_1, \\
 &(\alpha_4 - 1)k_1 - \alpha_4\}, \dots \\
 &\{k_1 - 1, k_0 - l_0, \alpha_3\}, \\
 &\{k_1 + l_0, -\alpha_1 + \alpha_4 + \alpha_3 - 1, \alpha_0 + 2, \\
 &(\alpha_1 + 1)l_0 + \alpha_4, \alpha_1 k_0 + (-\alpha_4 + 1)l_0, \\
 &(-l_0 - \alpha_4)k_0 + (-\alpha_4 + 1)l_0^2\}, \dots\} = 0
 \end{aligned}$$

$$\begin{aligned}
 D_8 = &\{\{k_1 + 1, k_0, \alpha_1 + 1, \alpha_4, \alpha_3 + 2, \alpha_0\}, \dots \\
 &\{k_1 - l_0, k_0, \alpha_0\}, \\
 &\{k_0 - l_0 k_1 + l_0^2, \alpha_1 + 1, k_1 - l_0 + \alpha_4 + 1, \\
 &k_1 - l_0 + \alpha_3 - 1, \alpha_0\}, \dots \\
 &\{k_1 - l_0, 2k_0 - l_0^2 - l_0, \alpha_1 + 2, l_0 + \alpha_4 + 2, \\
 &l_0 + \alpha_3 - 1, \alpha_0\}, \dots \\
 &\{k_0 - l_0 k_1 + l_0^2, \alpha_1 + 1, k_1 - l_0 + \alpha_4 + 1, \\
 &k_1 - l_0 - \alpha_3 - 1, \alpha_0\}, \dots \\
 &\{k_1 - l_0, 2k_0 - l_0^2 - l_0, \alpha_1 + 2, l_0 + \alpha_4 + 2, \\
 &l_0 - \alpha_3 - 1, \alpha_0\}, \dots \\
 &\{k_1, k_0 - l_0, \alpha_1 + 1, \alpha_4 + 2, \alpha_3, \alpha_0\}, \dots\} = 0
 \end{aligned}$$

$$\begin{aligned}
 D_9 = &\{\{l_0, k_1, \alpha_3 - 1, \alpha_1 - \alpha_4 - \alpha_0, \alpha_1 k_2 - \alpha_4\}, \\
 &\dots \{l_0, k_2 - 1, k_1, \alpha_0\}, \dots \\
 &\{2l_0 - 1, 2k_2 - 1, k_1 - 1, \alpha_1 - 2, \alpha_4 + 2, \alpha_3 - 2, \\
 &\alpha_0 - 1\}, \dots \\
 &\{2l_0 - 1, 2k_2 + 1, k_1 + 1, \alpha_1 - 2, \alpha_4 + 2, \alpha_3 - 2, \\
 &\alpha_0 - 3\}, \dots \\
 &\{k_2 - 1, k_1 - l_0, \alpha_1 l_0 - \alpha_1 + \alpha_4, \\
 &\alpha_0, l_0^2 - l_0 + 1, \alpha_1^2 l_0 - \alpha_1^2 + \alpha_3^2 + 1\}, \\
 &\{k_2 - l_0, k_1 - l_0 + 1, \alpha_1 l_0 - \alpha_4, \alpha_3 - 1, \\
 &\alpha_1 l_0 - \alpha_1 + \alpha_0, l_0^2 - l_0 + 1\}, \dots \\
 &\{k_2 - 1, k_1 - l_0, \alpha_0, l_0^2 - l_0 + 1, \\
 &\alpha_1^2 l_0 - \alpha_1^2 + \alpha_3^2 + 1\}, \\
 &\{k_2 - 1, k_1 - l_0, \alpha_0, l_0^2 - l_0 + 1\}\} = 0
 \end{aligned}$$

$$\begin{aligned}
 D_{10} = &\{\{k_2, k_1, \alpha_4\}, \\
 &\{k_1 - l_0 k_2, \alpha_3 + 1, \alpha_1 + \alpha_4 + \alpha_0, \alpha_1 k_2 + \alpha_4\}, \dots \\
 &\{k_2 - 1, k_1 - l_0, \alpha_0\}, \\
 &\{k_2, \alpha_1 + \alpha_4 + \alpha_3, \alpha_0 + 1, \alpha_1 k_1 + \alpha_4\}, \dots \\
 &\{k_1 + (-l_0 + 1)k_2 - 1, \alpha_4 + 1, \\
 &\alpha_1 + \alpha_3 + \alpha_0, \alpha_1 k_2 + \alpha_3\}, \dots \\
 &\{\alpha_1 + 1, k_1 + (-l_0 + 1)k_2 + \alpha_4, k_1 - l_0 k_2 + \alpha_3 - 1, \\
 &k_2 + \alpha_0 - 1\}, \dots \\
 &\{k_1 + (-l_0 + 1)k_2 - 1, \alpha_4 + 1, \\
 &\alpha_1 + \alpha_3 - \alpha_0, -\alpha_1 k_2 - \alpha_3\}, \dots \\
 &\{\alpha_1 + 1, k_1 + (-l_0 + 1)k_2 + \alpha_4, k_1 - l_0 k_2 + \alpha_3 - 1, \\
 &k_2 - \alpha_0 - 1\}, \dots \\
 &\{k_2, \alpha_1 + \alpha_4 - \alpha_3, \alpha_0 + 1, -\alpha_1 k_1 - \alpha_4\}, \dots \\
 &\{k_2, k_1 - 1, \alpha_3\}, \\
 &\{k_1 + (-l_0 + 1)k_2 - 1, \alpha_4 + 1, \alpha_1 - \alpha_3 + \alpha_0, \\
 &\alpha_1 k_2 - \alpha_3\}, \dots \\
 &\{\alpha_1 + 1, k_1 + (-l_0 + 1)k_2 + \alpha_4, k_1 - l_0 k_2 - \alpha_3 - 1, \\
 &k_2 + \alpha_0 - 1\}, \dots \\
 &\{k_1 + (-l_0 + 1)k_2 - 1, \alpha_4 + 1, \alpha_1 - \alpha_3 - \alpha_0, \\
 &-\alpha_1 k_2 + \alpha_3\}, \dots \\
 &\{\alpha_1 + 1, k_1 + (-l_0 + 1)k_2 + \alpha_4, k_1 - l_0 k_2 - \alpha_3 - 1, \\
 &k_2 - \alpha_0 - 1\}, \dots\} = 0
 \end{aligned}$$

7. Conclusion

We proposed the technique for obtaining solutions to ordinary differential equations. And, we also demonstrated the implementation of this technique and showed its timing data. If we do not use the resultant-factorization technique, we cannot obtain all the solutions for the ideal (2) within 48 h. However, we can compute solutions with Table 1 by using resultant-factorization technique.

References

- [1] Y. Kawano, K. Kimura, H. Sekigawa, M. Noro, K. Shirayanagi, M. Kitagawa, and M. Ozawa. Existence of the exact CNOT on a quantum computer with the exchange interaction. *Quantum Inf. Process.*, 4(2):65–85, 2005.
- [2] K. Kajiwara, T. Masuda, M. Noumi, Y. Ohta, and Y. Yamada. Determinant formulas for the toda and discrete toda equations. *Funkcial. Ekvac.*, 44:291–307, 2001.
- [3] M. Mazzocco. Rational solutions of the painlevé vi equation. *J. Phys. A: Math. Gen.*, 34:2281–2294, 2001.
- [4] T. Shimoyama and K. Yokoyama. Localization and primary decomposition of polynomial ideals. *J. Symbolic Comput.*, 22(3):247–277, 1996.
- [5] T. Sturm and A. Weber. Investigating genetic methods to solve Hopf bifurcation problems in algebraic biology. In K. Horimoto, G. Regensburger, M. Rosenkranz, and H. Yoshida, editor, *Algebraic Biology*, volume 5147 of *Lecture Notes in Computer Science*, pages 200–215. Springer(Heidelberg), 2008.
- [6] H. Yoshida, K. Kimura, N. Yoshida, J. Tanaka and Y. Miwa Algebraic approaches to underdetermined experiments in biology. *IPSJ Transactions on Bioinformatics*, 3:62–69, 2010.

Hierarchical Visualization of Similarities between Probabilistic Distributions for Profiling

Akira Ito, Tomohiro Yoshikawa, Takeshi Furuhashi

Dept. of Computational Science and Engineering
Nagoya University
Furo-cho, Chikusa-ku, Nagoya 466-8603, Japan

Abstract—*One of the most important purposes in the analysis of questionnaire data is to get profiles for the target group(s). As the result of profiling gives a great influence on the planning marketing strategy, the reliability of profiling is very important. Correspondence Analysis, Association Rule mining are typical methods for profiling and are useful to grasp the relationships between categorical variables in data. However, the results of these methods may “overfit” to the data and be different from the behavior in the general population, especially when the sample size is small. This paper proposes a new profiling method considering behavior in the general population by the probabilities. It derives the probabilistic distributions of each choice on a question by Bayesian approach for each attribute, and visualizes the similarities between these distributions. This paper applies the proposed method to an actual questionnaire data on a scanner product. It shows that a user can profile the data considering the uncertainty of extracted rules for the relationship between each attribute and the answer to the essential question. It also shows that the visualization supports a user to grasp the similarities between the probabilistic distributions for each attribute and to extract the characteristics of attributes.*

Keywords: Analysis of Questionnaire Data, Profiling, HPD Interval, Similarity between Probabilistic Distributions, Hierarchical Visualization

1. Introduction

In the marketing field, questionnaires are often carried out and the acquired data are analyzed in order to grasp the market trends and to plan a marketing strategy. For example, when a company plans a marketing strategy for a new product, it often surveys the impression of customers for the product by a questionnaire. It analyzes the questionnaire data and utilizes the results for the prediction of marketing scale or the decision of the target groups to sell[1][2].

One of the most important purposes in the analysis of questionnaire data is to find “Who are the target group?,” in other words, to get profiles for the target group(s). As the result of profiling gives a great influence on the planning marketing strategy, the reliability of profiling is

very important. Various methods for profiling have been employed so far.

Correspondence Analysis[3], Association Rule mining[4] are typical methods for profiling[5][6]. Correspondence Analysis is a technique designed to analyze two-way or multi-way contingency tables. The goal of a typical analysis is to represent the relationship between the rows and/or columns of the table, e.g., hobbies and ages, on the visualized space. Association Rule mining is that to extract the regularities in data, and the analysis is based on the evaluation indices for association rules. The evaluation indices for the rule $\{A \Rightarrow B\}$ represent the relationship between A and B, for example, one of them represents the conditional probability of having item B given item A. These methods are useful to grasp the relationships between categorical variables in data. However, the results of these methods may “overfit” to the data and be different from the behavior in the general population, especially when the sample size is small[7][8]. It may lead to unreliable profiling.

This paper proposes a new profiling method considering behavior in the general population by the probabilities. It assumes that people with attribute X respond to each choice on a question with the probability θ . For each attribute, the probabilistic distribution of the question with the choices is derived by Bayesian approach, and the expectation value and Highest Posterior Density (HPD) interval[9] of the distribution are defined as the evaluation indices. Then, each attribute is mapped onto the visualized space based on the similarities between the probabilistic distributions for them.

This paper applies the proposed method to an actual questionnaire data on a scanner product. It shows that a user can profile the data considering the uncertainty of extracted rules for the relationship between each attribute and the answer to the essential question with the derivation of their probabilistic distributions and evaluation indices. It also shows that the visualization supports a user to grasp the similarities between the probabilistic distributions for each attribute and extract the characteristics of attributes.

2. Proposed Method

It assumes that people with attribute X respond to a question with the probabilities for each choice $\theta =$

$\{\theta_1, \theta_2, \dots, \theta_k\}$ (see Fig.1). These probabilities are the parameters and the probabilistic distribution of them is derived by Bayesian approach. The expectation value and HPD interval of the distribution are defined as the evaluation indices. The probabilistic distribution for every attribute on the question is derived and the similarities between them are calculated. Then, each attribute is mapped onto the visualized space based on these similarities.

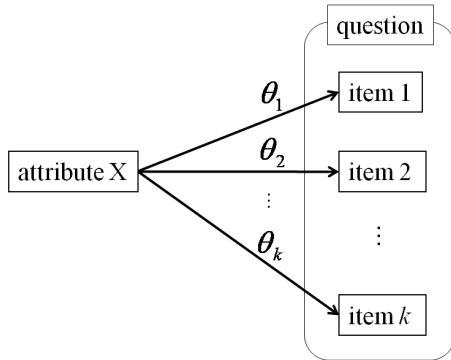


Fig. 1: Answer to question with k choices by respondents with attribute X

2.1 Derivation of Probabilistic Distribution and Evaluation Indices

2.1.1 Probabilistic Distribution

Let $\theta = \{\theta_1, \theta_2, \dots, \theta_k\}$ denote a set of parameters and $\mathbf{x} = \{x_1, x_2, \dots, x_k\}$ denote the data. θ_i represents the probability of choice i to a question and x_i represents the number of respondents who chose item i from a set of k items (choices). The probability of the data \mathbf{x} under the parameters θ is given by

$$p(\mathbf{x}|\theta) = \frac{(\sum_i x_i)!}{\prod_i x_i!} \prod_{i=1}^k \theta_i^{x_i}. \quad (1)$$

The probabilistic distribution of the parameters is derived from the data by Bayes' theorem.

$$p(\theta|\mathbf{x}) = \frac{p(\mathbf{x}|\theta)p(\theta)}{p(\mathbf{x})} \propto p(\mathbf{x}|\theta)p(\theta) \quad (2)$$

$p(\theta|\mathbf{x})$ is the posterior distribution and $p(\theta)$ is the prior distribution. Now, we set the prior distribution as the eq.(3) below.

$$p(\theta; \alpha) = \frac{\Gamma(\sum_i \alpha_i)}{\prod_i \Gamma \alpha_i} \prod_{i=1}^k \theta_i^{\alpha_i}. \quad (3)$$

$p(\theta; \alpha)$ is a conjugate prior for the likelihood (eq.(1)). $\alpha = \{\alpha_1, \alpha_2, \dots, \alpha_k\}$ is called hyper parameters. From eq.(1), (2)

and (3), the posterior distribution is given by

$$\begin{aligned} p(\theta|\mathbf{x}) &= \frac{\Gamma(\sum_i (x_i + \alpha_i))}{\prod_i \Gamma(x_i + \alpha_i)} \prod_{i=1}^k \theta_i^{x_i + \alpha_i - 1} \\ &= \frac{1}{B(\mathbf{x} + \alpha)} \prod_{i=1}^k \theta_i^{x_i + \alpha_i - 1}. \end{aligned} \quad (4)$$

This distribution form is called Dirichlet distribution[10].

2.1.2 Evaluation Indices

The expectation value (mean) and the HPD interval of the posterior distribution are defined as the evaluation indices for each attribute.

Expectation value

The expectation value of the Dirichlet distribution is analytically derived[10]. In the case of eq.(4), the expectation value is given by

$$E[\theta_i] = \int p(\theta|\mathbf{x}) \theta_i d\theta = \frac{x_i + \alpha_i}{\sum_i (x_i + \alpha_i)}. \quad (5)$$

HPD interval

An interval R in the parameter space is called 100(1- ϵ)% HPD interval, and it has the following characteristics.

1. $p(\theta \in R|\mathbf{x}) = 1 - \epsilon$
2. for $\theta_1 \in R$ and $\theta_1 \notin R$, $p(\theta_1|\mathbf{x}) \geq p(\theta_2|\mathbf{x})$

The HPD interval has k dimensions and it can be obtained by numerical computing[9].

Fig.2 shows an example of these evaluation indices. The larger sample size is, the closer the expectation value becomes to mode (simple conditional probability in data) and the smaller the HPD interval becomes.

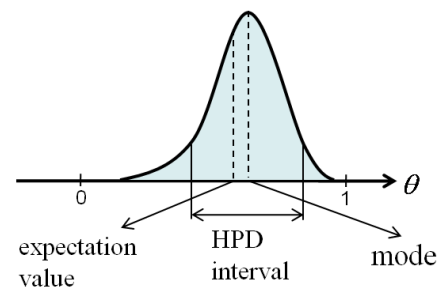


Fig. 2: Example of evaluation indices

2.2 Visualization of Similarities between Probabilistic Distributions

2.2.1 Similarities between Probabilistic Distributions

This paper employs the Bhattacharyya distance as the measure of the similarity between probabilistic distributions.

The Bhattacharyya distance J_B between two Dirichlet distributions,

$$p_a = \frac{1}{B(\lambda_a)} \prod_i \theta_i^{\lambda_{ai}-1}, \lambda_a = \{\lambda_{a1}, \lambda_{a2}, \dots, \lambda_{ak}\} \text{ and}$$

$$p_b = \frac{1}{B(\lambda_b)} \prod_i \theta_i^{\lambda_{bi}-1}, \lambda_b = \{\lambda_{b1}, \lambda_{b2}, \dots, \lambda_{bk}\},$$

is analytically derived as the eq.(6) below[11].

$$J_B = -\ln \int_{\theta} \sqrt{p_a p_b} d\theta = \ln \frac{\sqrt{B(\lambda_a)B(\lambda_b)}}{B(\frac{1}{2}\lambda_a + \frac{1}{2}\lambda_b)} \quad (6)$$

2.2.2 Multi Dimensional Scaling with Spring Model

Multi Dimensional Scaling method[12] can visualize the similarities (distances) between instances in the original space. The distance between instances i and j in the original space and that in the visible space are defined as d_{ij} and d_{ij}^* , respectively. The coordinates of instances in the visible space, which are determined randomly at the beginning, can be calculated by minimizing the energy function E .

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^n k_{ij} (d_{ij} - d_{ij}^*)^2 \quad (7)$$

k_{ij} is a control parameter, which is called a spring coefficient. The distance between the two instances in the visible space gets close to be equal to that in the original space when k_{ij} becomes larger.

2.2.3 Hierarchical Visualization with Segmentation of Attributes

Each attribute is mapped onto the visible space by Multi Dimensional Scaling method with spring model described above based on the similarities between the probabilistic distributions for the attribute. Fig.3 shows the image of the visualization of attributes.

First, the visualization of rank 0 and rank 1 is done (see Fig.3(a)). Rank0-attribute means the probabilistic distribution for all data and rank1-attribute means that for a single attribute such as {male}, {20's}, and so on. They are mapped together onto the same visible space while the spring coefficients between the different ranks are set to be large value to keep their original distances. Next, a user can see the relationship in the lower rank for more detail analysis. He/she chooses an attribute and then the visualization with the segmentation of the attribute is done in the same way to the rank 0-1 visualization (see Fig.3(b)). Rank2-attribute means the combination between the chosen attribute and others such as {male & 30's}, {male & part-timer}, and so on when the attribute {male} is chosen. This hierarchical visualization enables the user to intuitively grasp the similarities of the attributes in terms of the probabilistic distributions on the question between the different ranks and in the same rank, and to extract the characteristics of attributes for profiling.

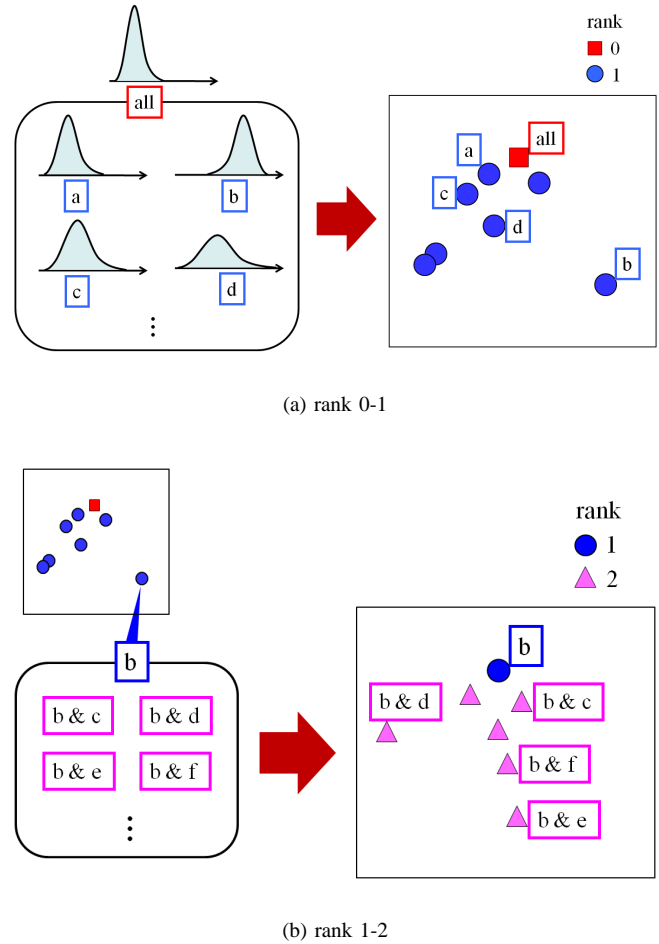


Fig. 3: Image of visualization of attributes

3. Experiment and Discussion

3.1 Experimental Setting

In this experiment, questionnaire data on a scanner product was used. The number of respondents was 1564. We focused on the question “Utilization experience of scanner” (Did you use scanner(s) in the past year?). TABLE 1 shows the questions related to respondents’ attributes.

Table 1: Questions on attributes

Contents
Sex
Age
Occupation
Household income

3.2 Comparison with Association Rule Mining

In this section, we compare the evaluation indices of the proposed method, the expectation value and the HPD interval, with the *confidence*, which is one of the main evaluation indices in Association Rule[4][6], in terms of their estimation errors.

Association Rule mining is that to extract the regularities in data, and the evaluation indices for a rule $\{A \Rightarrow B\}$ represent the relationship between A and B. *Support* represents the probability of having both item A and item B. *Confidence* represents the conditional probability of having item B given item A.

The hyper parameter α_i was set to 1 (uniform prior distribution), and this experiment was done as follows:

- 1) 100 samples D_s were randomly extracted from all data D , 1564 samples, in the questionnaire data.
- 2) The rules R whose consequent was {Utilization experience of scanner = no} in D_s under the conditions, $support \geq a$ and $rank \leq 2$, were extracted. R_{10} denotes the rules which have top 10 *confidence* in R .
- 3) $E1$ was evaluated, which is the average of the difference between *confidence* on D_s and *confidence* on D for R_{10} .
- 4) $E2$ was evaluated, which is the average of the difference between expectation value on D_s and *confidence* on D for R_{10} .
- 5) $E3$ was evaluated, which is the proportion that *confidence* on D was out of 95% HPD interval on D_s for R_{10} .

Fig.4 shows the result of the comparison. The horizontal axis is a , the threshold of *support*. *Confidence-error* represents the average of $E1$ in 10 trials and expectation-error, HPD-error represents that of $E2$, $E3$. The lower a becomes, the larger *confidence-error* is while expectation-error is relatively small in any a . HPD-error for 95% HPD interval is also small, which is lower than or equal to 5%. Although we set the condition $rank \leq 2$ in this experiment, it seems that *confidence-error* becomes larger when we accept higher-rank rules because these rules generally have higher *confidence* but lower *support*.

The result shows that the expectation value gives better estimation than *confidence* which is the evaluation index for closed data, and we can also consider the reliability of rules by HPD interval. It seems that the evaluation indices of the proposed method are suitable for the profiling in the analysis of questionnaire data especially when the sample size is small.

3.3 Profiling

Fig.5 shows the visualization results by the proposed method and TABLE 2 shows the evaluation indices, expectation value and 95% HPD interval, for some attributes in the

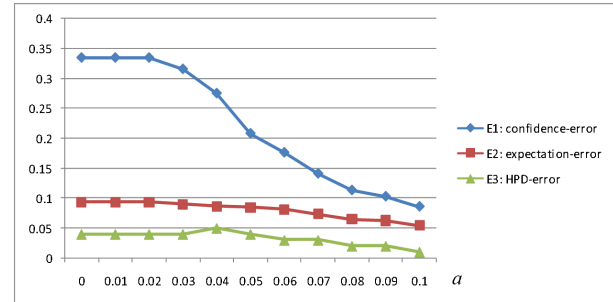


Fig. 4: Estimation error

figure when the hyper parameter α_i was set to 1 (uniform prior distribution). ID A-F in TABLE 2 correspond to those in Fig.5.

Fig.5 shows the result of rank 0-1 visualization. A, B and C are plotted far from the rank 0, all data. From Fig.5(a) and TABLE 2, it seems that all respondents tend not to use scanner(s) frequently. {Occupation = homemaker}, {Occupation = part-timer} are the groups which tend to use scanners less frequently than all respondents and {Household income = over 20 million yen}, {Occupation = employee (executive)} are those to use more frequently. There is no overlap in HPD interval between these attributes and all respondents. So, these groups have significantly different trends from all respondents in the utilization experience of a scanner.

We focused on the attributes {Occupation = employee (executive)}, {Household income = over 20 million yen} and more analysis was done by seeing the lower rank of them. Fig.5(b) shows the result when {Occupation = employee (executive)} was segmented. The expectation value of {Utilization experience of scanner = yes} for {Occupation = employee (executive) & Household income = 8-10 million yen} is lower and that for {Occupation = employee (executive) & Household income = 18-20 million yen} is higher than that for the higher rank {Occupation = employee (executive)}. From these results, the amount of household income is related to the utilization experience of a scanner in those who are executive-employees. However, the HPD interval for {Occupation = employee (executive) & Household income = 18-20 million yen} is large and the lower bound is 0.23. It means that this rule or profile is uncertain because the number of applicable data is not enough.

Fig.5(c) shows the result when {Household income = over 20 million yen} was segmented and F is the part of them at rank 2. The variance of the attributes is very small in Fig.5(c). It means that the probabilistic distributions for these attributes obtained by the segmentation of {Household income = over 20 million yen} are similar one another. In these attributes, {Household income = over 20 million yen} is strongly affected and there are no significant differences

Table 2: Evaluation indices

ID	Attribute	Utilization experience of scanner = no	Utilization experience of scanner = yes
	all	0.79 (0.77-0.81)	0.21 (0.19-0.23)
A	Occupation = homemaker	0.87 (0.83-0.91)	0.13 (0.09-0.17)
	Occupation = part-timer	0.90 (0.83-0.95)	0.10 (0.05-0.17)
B	Household income = over 20 million yen	0.57 (0.41-0.73)	0.43(0.27-0.59)
C	Occupation = employee (executive)	0.64 (0.57-0.71)	0.36 (0.29-0.43)
D	Occupation = employee (executive) & Household income = 8-10 million yen	0.75 (0.59-0.90)	0.25 (0.10-0.41)
E	Occupation = employee (executive) & Household income = 18-20 million yen	0.40 (0.04-0.77)	0.60 (0.23-0.96)
	Occupation = employee (executive) & Household income = unknown	0.40 (0.17-0.64)	0.60 (0.36-0.83)
F	Household income = over 20 million yen & Sex = male	0.58 (0.36-0.79)	0.42 (0.21-0.64)
	Household income = over 20 million yen & Age = 20's	0.57 (0.24-0.90)	0.43 (0.10-0.76)

※ expectation value (HPD interval)

of the trends in the utilization experience of a scanner among them.

4. Conclusion

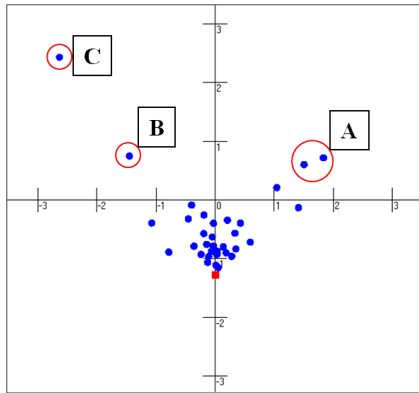
This paper proposed a new profiling method considering behavior in the general population by the probabilities. The proposed method derived the probabilistic distributions for each attribute of each choice on a question by Bayesian approach, and visualized the similarities between these distributions. The expectation value and HPD interval of the distribution were defined as the evaluation indices for the attributes. This paper applied the proposed method to an actual questionnaire data on a scanner product. It showed that a user could profile the data considering the uncertainty of extracted rules for the relationship between each attributes and the focused question. It also showed that the visualization supports a user to grasp the similarities between the probabilistic distributions for each attribute and the characteristics of attributes. Our future works are the study on the measures of the similarity between probabilistic distributions and the design of appropriate prior distribution (hyper parameters).

Acknowledgment

This work was supported in part by a Grant-in-Aid for Scientific Research (C) from the Ministry of Education, Culture, Sports, Science and Technology (MEXT), Japan, Grant number: 22500088.

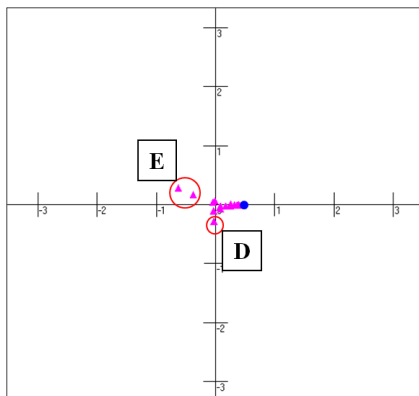
References

- [1] S. Liao, C. Chen, C. Hsieh, and S. Hsiao, "Mining information users' knowledge for one-to-one marketing on information appliance," *Expert Systems with Applications*, vol. 36, no. 3, pp. 4967-4979, 2009.
- [2] S. Kuroda, T. Yoshikawa, and T. Furuhashi, "A proposal for analysis of sd evaluation data by using clustering method focused on data distribution," in *Proc. of the International Symposium on Frontiers of Computational Science 2005, 2007*, pp. 317-320.
- [3] M. O. Hill, "Correspondence analysis: A neglected multivariate method," *Applied Statistics*, vol. 23, no. 3, pp. 340-354, 1974.
- [4] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. of the 20th VLDB Conf.*, 1994, pp. 487-499.
- [5] M. J. Greenacre, *Correspondence Analysis in Practice*. Chapman and Hall, London, 2007, vol. 23, no. 3.
- [6] J. Jiao and Y. Zhang, "Product portfolio identification based on association rule mining," *Computer-Aided Design*, vol. 37, no. 2, pp. 149-172, 2005.
- [7] X. Yin and J. Han, "Cpar: classification based on predictive association rules," in *Proc. of the 3th SIAM International Conf.*, 2003, pp. 331-335.
- [8] T. Scheffer, "Finding association rules that trade support optimally against confidence," *Intelligent Data Analysis*, vol. 9, no. 4, pp. 381-395, 2005.
- [9] M. Chen and Q. Shao, "Monte carlo estimation of bayesian credible and hpd intervals," *Computational and Graphical Statistics*, vol. 8, no. 1, pp. 69-92, 1999.
- [10] T. Minka, "Estimating a dirichlet distribution," M.I.T, Tech. Rep., 2000.
- [11] T. W. Rauber, T. Braun, and K. Berns, "Probabilistic distance measures of the dirichlet and beta distributions," *Pattern Recognition*, vol. 41, no. 2, pp. 637-645, 2008.
- [12] M. T. Pham, T. Yoshikawa, T. Furuhashi, and K. Tachibana, "Pattern recognition based on two-dimensional dendrogram map using spring model," in *Proc. of the 1th International Workshop on Aware Computing*, 2009, pp. 614-619.



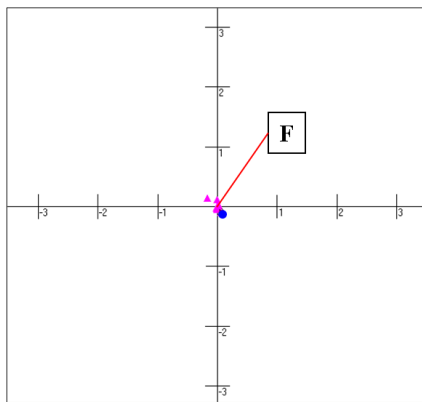
■ all

(a) rank 0-1



● Occupation =employee(executive)

(b) rank 1-2 (rank1-attribute:C)



● Household income=over 10 million yen

(c) rank 1-2 (rank1-attribute:B)

Fig. 5: Visualization result

Construction of a Mathematical Model and Quantitative Assessments of Impression in Western Painting

URANO Sachi¹

¹School of informatics, University of Tsukuba, Tsukuba, Ibaraki, Japan

Abstract - We present a new method to evaluate the impression of paintings, in terms of a mathematical modeling and quantitative assessments of impression in western painting. The present method is based on a detailed modeling of various factors and elements that consist of the composition of the painting. Impressions of 20 subject-people for 50 paintings are measured using the semantic differential method, which are compared to the impressions calculated with the present method. Their correlation is analyzed with Akaike's Information Criterion (AIC) standard. A strong correlation is obtained.

Keywords: composition, art, mathematical model, impression, parameter

1 Introduction

A number of fascinating paintings, which gives us a deep sympathy have remained up to now. These works are not only superior in their painting materials and techniques but also in the construction of the motif and the arrangement of the layout. There is an approach to understand the layout of objects in the painting by categorizing the whole screen to several common parts, which is called the composition.

Because the composition strongly affects the impression of the painting, it has been studied by many researchers to find a good composition or the general rule for the good painting, which gives a specific impression. In many previous studies and textbooks, it has been a common method to draw a parting line on the painting to analyze the composition. However, there are few works that succeeded to evaluate the relation between the impression and the composition, explicitly. Ozawa[1] analyzed the composition of *ukiyo*e paintings based on three-dimensional space geometry. Unfortunately it was difficult to expand the method to the western paintings. The research groups in Visual Perception and Aesthetics Lab[2] and Christopher Tyler Lab[3] tried to figure out the general rules for the composition in terms of an arrangement of the subject, using many photographs and paintings. While their approach was straightforward, the relation among the *composition elements* and their quantitative definition (*composition factors*) derived by their relation are not clearly expressed, since they focused on the intuitive and subjective appreciations of the viewers.

In this paper, we present a new mathematical approach to evaluate the relation between the composition and the impression, which is based on the analysis of a three-dimensional space geometry.

2 Description of the method

2.1 Terminology

The composition is classified into two major categories, which are the shape and the relation among the objects (*fundamental form*), and the size of the objects and their layout (*arrangement form*). We discuss the *arrangement form*, which can be analyzed mathematically. The following ten *composition factors* are prepared:

- 1) μ (*balance*), area fraction of elements on the left side of the screen after splitting the screen vertically in the middle.
- 2) ϕ (*density*), area fraction of all the elements that are in the screen.
- 3) J (*jump ratio*), area fraction of the smallest element to the biggest one inside the slit (magnification ratio).
- 4) X (*information value*), number of all elements.
- 5) σ_s (*slit ratio*), area fraction of the smallest rectangle that includes all the elements in a picture (*slit*) to that of the whole paintings.
- 6) p_+ (*upper vertical proportion*), range between the horizon and the upper limit of the picture.
- 7) p_- (*lower vertical proportion*), range between the horizon and the lower limit of the slit.
- 8) δ (*density parameter*), homothetic area of elements not to occlude each other.
- 9) λ (*similarity ratio*), homothetic ratio of the fronting element to the backend element.
- 10) σ_v (*printing domain vertical ratio*), slit height.

Objects in the painting are described with e_i $\{i = 1, 2, \dots, n\}$, where e_1 and e_n stand for the fronting and the backend elements, respectively.

2.2 Mathematical modeling and definition

2.2.1 Domains

We divided the screen into four *domains* to evaluate the *composition factors* and the object arrangement (Figure 1):

$$\begin{aligned}
 D_+ &= [-1, 1] \times [p_- + 2\sigma_v(1 + 2p_+), 2p_+] && \text{: upper domain} \\
 S &= [-1, 1] \times [-p_-, p_- + 2\sigma_v(1 + 2p_+)] && \text{: printing slit} \\
 D &\subset S && \text{: printing domain} \\
 D_- &= [-1, 1] \times [-1, -p_-] && \text{: lower domain}
 \end{aligned}$$

$$D_+ \cup S \cup D_- = [-1, 1] \times [-1, 2p_+] : \text{image plane.}$$

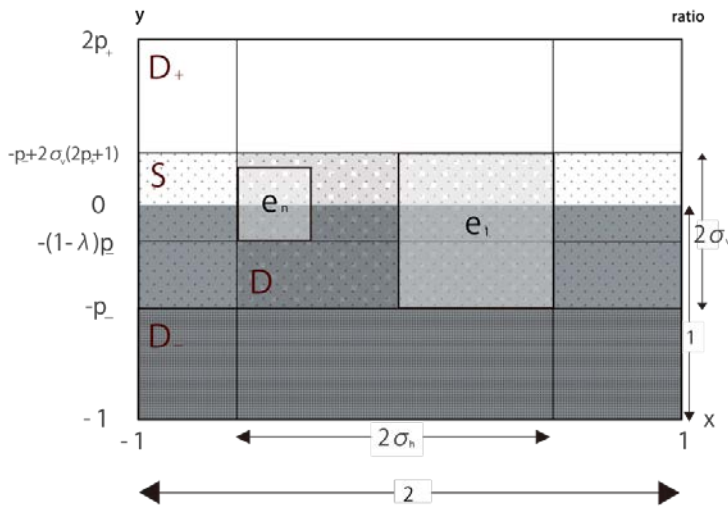


Figure 1: Domains and parameters (D_+ , S , D_- and D_- stand for upper domain, printing slit, printing domain and lower domain, respectively). $\{p_+, p_-, \sigma_v, \lambda\} \in [0, 1]$.

2.2.2 Object size and composition factors

A viewer maps objects in the three-dimensional space into the two-dimensional image plane through a screen settled between the objects and the viewer. It is done by fixing the observing point, which was followed by changing the visual points, several times.

Geometry models of the three different visual points towards a group of objects $\{O_i: i = 1, \dots, n\}$ that are lined up over the horizon in a similar space interval are shown in Figure 2 (objects are trees, in this example). We see how the objects are projected over the screen. We assume the same heights for the closest and the farthest trees, O_1 and O_n respectively, for the simplicity of the discussion. The heights of the visual points in the screen are set to: (a) the top of the trees, (b) between the top of the tree and the horizon and (c) the horizon, respectively. The image of the closest and the farthest trees e_1 and e_n aligns on (a) the top, (b) middle and (c) root of the trees, respectively, in the two dimensional plane. It is understood that the size and layout of the images $\{e_n\}$ in two-dimensional picture plane are the function of the relative position of the horizon and the slit; range between the horizon and the upper limit of the picture (p_+), the lower limit of the slit (p_-) and the slit height (σ_v), respectively, when we

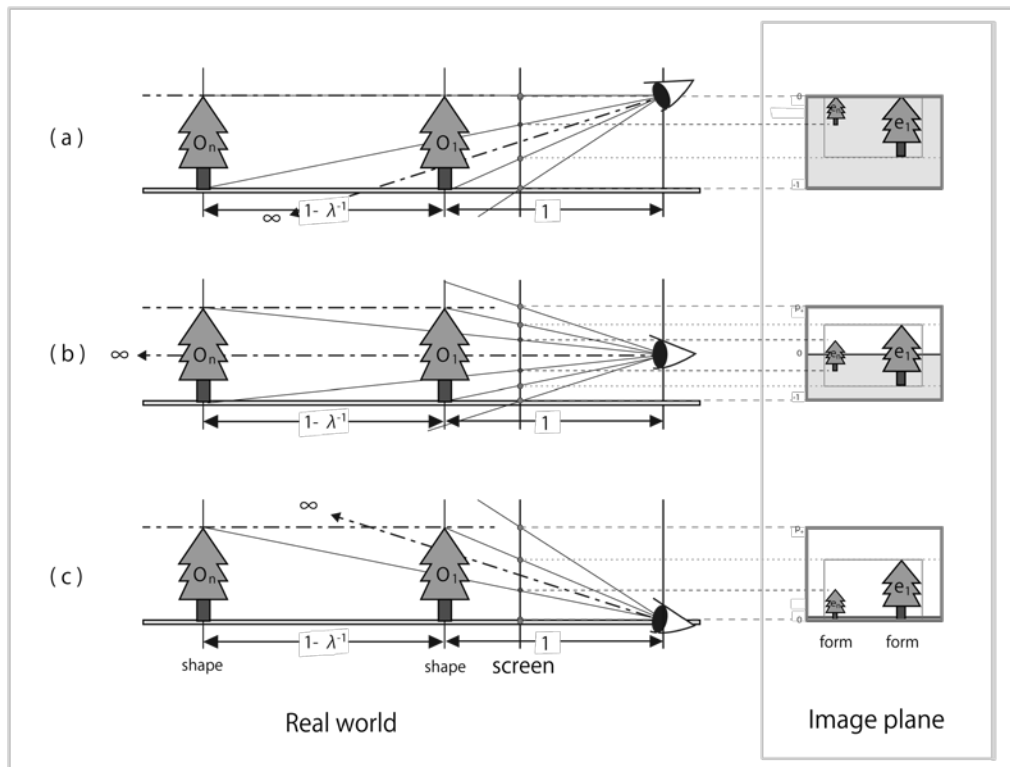


Figure 2: Geometry models of the three different visual points towards a group of trees (a: top of the trees, b: between the top of the trees and the horizon, c: the horizon).

observe the objects from the fixed visual point.

The area ratio of e_1 to e_n is λ^2 , where λ is a homothetic ratio of e_1 to e_n . The distance between O_1 and O_n is $1 - \lambda^{-1}$ when we assume the distance between the viewer and O_1 of 1. The *composition elements* are expressed as follows,

$$\begin{aligned} \{e_i \subset D: i = 1, 2, \dots, n\}: & \text{ texture elements} \\ e_1 \subset D & \quad \quad \quad : \text{ fronting element} \\ e_n \subset D & \quad \quad \quad : \text{ backend element,} \end{aligned}$$

using the following *parameters*

$$\begin{aligned} \rho & \in [0, 1] & : \text{ principal and subsidiary element ratio} \\ \rho_p & \in [0, 1] & : \text{ principal elemental ratio} \\ \rho_s & \in [0, 1] & : \text{ subsidiary elemental ratio} \\ \delta & \in [0, 1] & : \text{ density parameter} \\ \varepsilon & & : \text{ quantization unit of } e_n. \end{aligned}$$

When we introduce the *secondary parameters*,

$$\begin{aligned} \sigma_h & \in [0, 1] & : \text{ printing domain horizontal ratio,} \\ n & \in [1, 2, \dots, N] & : \text{ information number,} \end{aligned}$$

we obtain,

$$\begin{aligned} \sigma_s & = \frac{2\sigma_h\sigma_v}{1+2p_+} \in [0, 1] & : \text{ slit ratio,} \\ J & = (1 - \lambda)^2 \in [0, 1] & : \text{ jump ratio (reciprocal),} \\ \chi & = \frac{n}{N} \in [0, 1] & : \text{ information value.} \end{aligned}$$

2.2.3 Element layout rule

We consider a digitized picture or painting. The smallest element size ε is $\min(\frac{2}{H}, \frac{1+2p_+}{V})$, when the screen has $H \times V$ pixels. Assuming the heights of the fronting and backend elements of d and $(1 - \lambda)d$ respectively, we have the following equations:

$$\text{(Case 1)} \quad p_+ \geq \frac{\varepsilon}{4\sigma_v(1-\lambda)} - \frac{1}{2}, \text{ in case of } p_- \leq 2\sigma_v(1 + 2p_+), \quad (1)$$

$$\text{(Case 2)} \quad p_+ \geq \frac{\varepsilon + \lambda p_-}{4\sigma_v} - \frac{1}{2}, \text{ in case of } p_- > 2\sigma_v(1 + 2p_+). \quad (2)$$

When we choose p_+ so as to satisfy the above mentioned conditions, the height d is uniquely expressed with the four fundamental parameters $\{p_+, p_-, \sigma_v, \lambda\}$:

$$\text{(Case 1)} \quad d = 2\sigma_v(1 + 2p_+), \quad (3)$$

$$\text{(Case 2)} \quad d = \frac{2\sigma_v(1+2p_+) + \lambda p_-}{(1-\lambda)}. \quad (4)$$

Figure 3 shows the two types of arrangement models. In *case 1*, the fronting element e_1 is the largest in the picture, since it is inscribed inside S , bordering upper and lower parts. In *case 2*, e_1 is not the largest element, since e_1 and e_n border S on the lower and upper sides, respectively, yet both are inscribed in S .

The height of the elements $\{e_i: i = 2, \dots, n - 1\}$ becomes $-\frac{d}{p_-}y$, when we assume the base coordinates of $y \in [-p_-, -(1 - \lambda)p_-]$, because of the mutual similarity.

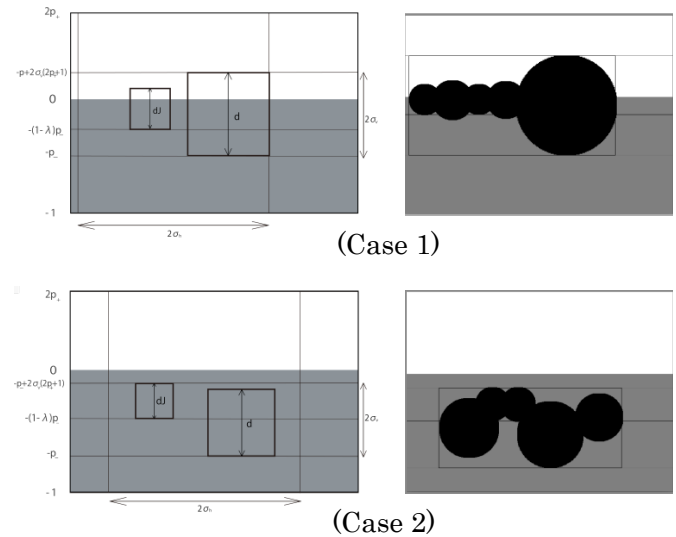


Figure 3: Arrangement of the elements (a : Case 1, b : Case 2).

We assume $\{e_1, e_n\}$ of the principal elements, and the proportion of the height and width of 1: $4\rho_p$. The height ratio of the principal and subsidiary elements is 1: ρ , so that the aspect ratio becomes $\rho: \rho\rho_s$. The elements $\{e_i: i = 2, \dots, n - 1\}$ can be principal or subsidiary (see Figure 4 for details).

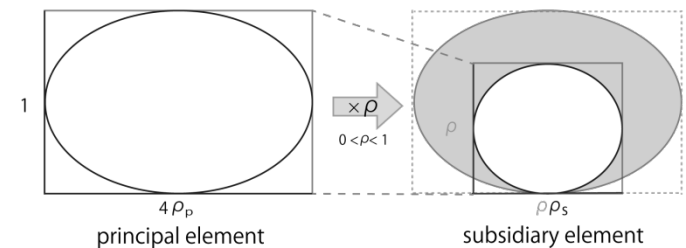


Figure 4: Principal and subsidiary elements.

All the elements that belong to the *printing slit S* must satisfy $e_i \subset S$. $\{e_1, e_n\}$ are arranged according to the *condition 1* explained below, to avoid occlusion $e_n \subset e_1$. Assuming the center of the element being fixed, we recursively arrange e_i in the paintings using the image of e_i , $U_\delta(e_i)$, whose homothetic ratio is 1: δ in order to fulfill both conditions below (see Figure 5 for details).

$$\text{(Condition 1)} \quad U_\delta(e_i) \cap (\bigcup_{j=1}^{i-1} e_j \cup e_n) = \emptyset \text{ for } \delta \in [0.5, 1.0], \quad (5)$$

$$\text{(Condition 2)} \quad e_i \cap (\bigcup_{j=1}^{i-1} e_j \cup e_n) \neq \emptyset \text{ for } \delta \in [0, 0.5]. \quad (6)$$

The images can be reduced or magnified depending on the size of δ :

reduced : $e_i \subset U_\delta(e_i)$ for $\delta \in [0.5, 1.0]$, (7)

magnified : $U_\delta(e_i) \subset e_i$ for $\delta \in [0, 0.5]$. (8)

If all the elements cannot be arranged, the number of the elements will be reduced to $i + 1$, where $i (< n)$ represents the number of the elements, which completed the arrangement.

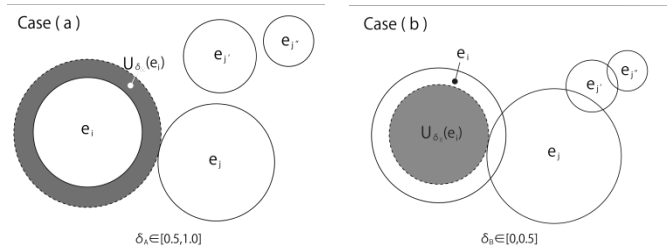


Figure 5: Elements layout rules(a: $e_i \subset U_\delta(e_i)$, b: $U_\delta(e_i) \subset e_i$).

3 Quantitative evaluation of impression

3.1 Sample preparation

An experimental confirmation about the present method has been carried out with an experiment. It is important to generate a geometric model that the objects in the paintings can be distinguished from the background, satisfying the conditions mentioned above, and that the picture is visually perspective with the horizon contained in it. We chose 50 western landscape paintings with people from the art books [4, 5, 6] that fulfill all the conditions. 44% paintings were from between 15th and 20th centuries, and 26% paintings were in 18th and 19th, which were categorized to the Romanticism, Neoclassicism, Realism and Impressionism. The paintings were processed into monochrome prior to the experiment. The maximum information number N was limited to ten.



Figure 6: Example of Principal and subsidiary elements.

In Figure 6, four people are pictured ($N = 4$). In this case, the objects $\{D, B\}$ or $\{e_1, e_4\}$ are the principal (fronting and the backend) elements. $\{e_2, e_3\}$ are the subsidiary elements with reduction ratios of $\rho = 0.6, \rho_s = 0.05, \rho_s = 0.38$, respectively. After arranging all the elements on the screen, we calculated the balance and the density, $\{\mu, \varphi\} \in [0, 1]$ [7].

model		parameter
		$\mu = 0.5873, \varphi = 0.0533$ $\lambda = 0.3025, \chi = 0.3000$ $\sigma_s = 0.2805, \rho_s = 0.850$ $\rho_s = 0.800, \delta = 0.600$ $\lambda = 0.450, \sigma_v = 0.500$
		$\mu = 0.0105, \varphi = 0.0347$ $\lambda = 0.0400, \chi = 0.6000$ $\sigma_s = 0.2137, \rho_s = 0.800$ $\rho_s = 0.800, \delta = 0.100$ $\lambda = 0.800, \sigma_v = 0.400$
		$\mu = 0.6364, \varphi = 0.2072$ $\lambda = 0.9025, \chi = 0.2000$ $\sigma_s = 0.2800, \rho_s = 0.750$ $\rho_s = 0.690, \delta = 0.100$ $\lambda = 0.050, \sigma_v = 0.700$

Figure 7: Examples of the paintings and the analytical results with the characteristic quantities.

3.2 Experimental

We evaluated the impressions of the paintings using the methods introduced by Ooyama [8]. The adjectives were categorized into three factors; evaluation factor, activity factor and potency factor, respectively. A rating scale group $\{i=1,2,\dots,10\}$, which consisted of ten pairs of adjectives of preferably independent meanings dispersed each of the factors was prepared to measure the impression. We showed 50 different paintings $\{j=1,2,\dots,50\}$ to the 20 people. The subject-people assessed their impressions using a scale of seven degrees (Semantic Differential Method).

We, then, calculated the mean impression $Y_{ij} \in [0, 1]$ for each painting. The impression according to the arrangement form $Im_{ij,e}$ was obtained,

$$Im_{ij,e} = Y_{ij} - \alpha_{ij}, \tag{9}$$

Table 1: Quantity evaluation of impression using AIC

Factor	No	Adjective	Function	AIC
Evaluation	1	Good (↔Bad)	$Im_{1j} = 0.358\varphi_{1j} - 0.355\chi_{1j} + 0.162\lambda_{1j} + 0.398$	-212.82
	2	Beautiful (↔Ugly)	$Im_{2j} = -0.454\chi_{2j} + 0.221\lambda_{2j} + 0.454$	-210.73
	3	Bright (↔Dark)	$Im_{3j} = 0.145\mu_{3j} - 0.119p_{+3j} - 0.463p_{-3j} + 0.16\lambda_{3j} + 0.777$	-183.17
	4	Noisy (↔Quiet)	$Im_{4j} = 0.272\chi_{4j} - 0.202\sigma_{v4j} + 0.648$	-180.91
Activity	5	Uncomfortable (↔Comfortable)	$Im_{5j} = 0.587$	-204.83
	6	Dynamic (↔Static)	$Im_{6j} = 0.358p_{-6j} - 0.332\sigma_{v6j} + 0.409$	-174.51
	7	Showy (↔Plain)	$Im_{7j} = -0.147\sigma_{v7j} + 0.601$	-212.87
	8	Unnatural (↔Natural)	$Im_{8j} = 0.403\chi_{8j} - 0.21\lambda_{8j} - 0.133\sigma_{v8j} + 0.6$	-200.15
Potency	9	Light (↔Heavy)	$Im_{9j} = 0.897\varphi_{9j} - 0.506\sigma_{s9j} + 0.124\delta_{9j} + 0.164\lambda_{9j} + 0.518$	-200.28
	10	Strong (↔Weak)	$Im_{10j} = 0.454$	-210.14

Table 2: Example of quantitative assessment of potency factor

Factor	No	Adjective	Function	AIC
Potency	9'	Light (↔Heavy)	$Im'_{9j} = 0.154\kappa_{9j} + 0.275p_{-9j} + 0.253$	-213.55
	10'	Strong (↔Weak)	$Im'_{10j} = -0.304p_{-10j} + 0.681$	-215.57

where, $|\alpha_{ij}| \in [0, 1]$ is the correction factor for the experiment uncertainty due to the impressions caused by the variously-shaped objects, eye direction and some patterns.

The impression calculated by the present method, $Im_{ij,c}$, can also be expressed using ten *composition factors*,

$$Im_{ij,c} = a_1^1\mu_{ij} + a_2^2\varphi_{ij} + a_3^3J_{ij} + a_4^4\chi_{ij} + a_5^5\sigma_{sij} + a_6^6p_{+ij} + a_7^7p_{-ij} + a_8^8\delta_{ij} + a_9^9\lambda_{ij} + a_{10}^{10}\sigma_{vij}, \quad (10)$$

where $\{a_k^k : k = 1, 2, \dots, 10\} \in [0, 1]$ are the free coefficients. We calculated the AIC-value, ($AIC = -2 \log(\text{maximum likelihood function}) - 2(\text{Arity})$) to evaluate the degree of correlation between the experimentally-obtained impression ($Im_{ij,e}$) and the calculated one by the composition factors ($Im_{ij,c}$). The multiple classification analysis was applied to evaluate the correlation. Table 1 summarizes the adjectives and the impression, Im_{ij} , which gave the minimum AIC-values. According to the AIC standard, the smallest AIC-value represents the strongest correlation.

3.3 Discussion

Im_{ij} shows the impression in numerical form. Multiple parameters were required to evaluate the impression except for adjectives No.5 and No.10. Here, we can observe a correlation between the impression obtained by the 20 subject-people and that evaluated with the present method.

We have found that people feel the painting noisy, when the impression was expressed with *information value* χ_{4j} and *printing domain vertical ratio* σ_{v4j} . It has also been pointed out by the previous study carried out by the ref.[7].

The definition of the *composition factors* affect to the evaluation results. For example, if we define $\kappa = \{2\mu \text{ for } \mu \in [0, 0.5], -2\mu + 2 \text{ for } \mu \in [0.5, 1.0]\}$, and analyze

$$Im'_{ij} = a_1^1\kappa_{ij} + a_2^2J_{ij} + a_3^3\chi_{ij} + a_4^4\sigma_{sij} + a_5^5p_{+ij} + a_6^6p_{-ij} + a_7^7\sigma_{vij}, \quad (11)$$

the potency factors will change (Table 2), and the AIC-values are even smaller.

We may have to introduce other *composition factors*, or appropriate *factors* other than the composition to improve the strength of the correlation, so that it well represents the impression of the people. The selection of the adjectives also affects the results. We will survey various adjectives that show strong correlation to the composition.

4 Conclusions

In summary, we were able to explicitly evaluate the impression of the people for the 50 different paintings, with the present method, which was based on the detailed evaluation of the *composition factors* of the paintings. The results obtained with the present study supported that the impression can be quantitatively assessed. We will improve the present method by introducing appropriate

composition factors and other ones to find the strong correlation to the impression. The present method can be applied to the evaluation of the impression not only for the paintings, but also for the photographs. It may also be applied to the other scientific fields, such as Kansei Engineering.

5 Acknowledgement

The author thank Prof. K. Sakai from the University of Tsukuba for his valuable comments. This work is supported in part by MEXT "Support project for students majoring in the field of math and science (University of Tsukuba)".

6 References

- [1]Ozawa Kazumasa: Towards a Model of Painting – A Consideration, Information Processing Society of Japan, 1996.
- [2]Jonathan S. Gardner et al.: Exploring Aesthetic Principles of Spatial Composition Through Stock Photography, VSS Poster, 2008.
- [3]CHRISTOPHER W. TYLER: Some principles of spatial organization in art, Spatial Vision Vol.20 No.6, pp.509-530, 2007.
- [4]John Oliver Hand: National Gallery of Art Master Paintings from the Collection, ABRAMS, 2004.
- [5]Vincent Pomerode: 1001 Paintings at the Louvre, MUSEE DU LOUVRE EDITIONS, 2008.
- [6]Stephan Farthing: 1001 Paintings You must See Before You Die, UNIVERSE, 2007.
- [7]VISUAL DESIGN LABORATORY INC.: Basics of Composition(Japanese), VISUAL D.
- [8]Ooyama Tadasu et al.: Chapter 4, Means for Psychological Investigating, SAIENSU-SHA, pp.65-75(Japanese), 2005.

Abstraction of DNA Graph Structures for Efficient Enumeration and Simulation

Ibuki Kawamata, Fumiaki Tanaka, Masami Hagiya

Department of Computer Science,

Graduate School of Information Science and Technology, University of Tokyo

ibuki@is.s.u-tokyo.ac.jp, fumi95@is.s.u-tokyo.ac.jp, hagiya@is.s.u-tokyo.ac.jp

Abstract—We propose a graph model of DNA molecules and an abstraction of that model for efficient simulation of molecular systems powered by DNA hybridization. In this paper, we first explain our DNA molecule model composed of graph data structures and highlight the problem of the large number of DNA structures that results. We then define an abstraction of the model, which focuses on local structures of DNA strands, and introduce reactions among the local structures. To verify the effectiveness of the abstraction, we develop simulators for the original and abstract models, and compare the number of structures generated by those simulators. Based on this research, computer-aided design of reaction systems that consist of biological molecules may become easier than conventional designs that rely on human trial and error.

Keywords: DNA computing, graph, local structure, simulation, enumeration

1. Introduction

Molecular systems using DNA and its simple hybridization mechanism have been recently developed, including nano-scale DNA structures [1], [2], DNA logic gates [3], [4], and DNA amplification machines [5], [6]. The design of such systems, however, is extremely difficult for humans because the combination of molecules in the system increases rapidly as the number of molecular species increases. This combinatorial explosion prevents humans from predicting system behavior and limits the total number of molecular species that can be used.

A variety of approaches for overcoming this difficulty in combining molecules have been proposed, most of which are based on simplified molecules and restricted reactions. Good examples of such approaches include simple hairpin strands of DNA that allow a cascade of reactions [7], the programming language for DNA circuits [8], and the computer-aided tool to produce three-dimensional DNA structures [9]. Even these methods, however, still require human trial and error to synthesize systems of interest. These researches simplified and restricted DNA structures and reactions so that systems to be designed were limited in their functions.

We previously proposed a method for the automatic design of DNA logic gates to synthesize small systems based on a flexible DNA model without human trial and error [10]. In that method, we defined a graph data structure to represent DNA molecules and developed a simulator based on chemical

kinetics. Although we restricted the model of DNA structures and introduced threshold to ignore unimportant structures, the simulator still led to a combinatorial explosion of structures.

In this study, we abstract the model by focusing on the local structure of DNA strands to overcome the explosion problem. The approach based on the local structures is similar to the equilibrium computation for hybridization reaction systems [11] and the rule-based language for cellular signaling pathways [12].

The organization of this paper is as follows. Section 2 reviews our previous work which introduced the graph model of DNA and three reactions among graph structures. Section 3 illustrates the unbounded increase of structures initiated by three types of molecules, which is an instance of the main problem in simulating DNA hybridization systems. Section 4 is the main contribution of the paper, which gives an abstraction of graph data structures. Section 5 explains how we simulate DNA hybridization systems based on our original and abstracted models. Section 6 exhibits experimental results, which verify the advantage of our work. Section 7 and 8 are the discussion and conclusion of this paper.

2. Graph Structure Modeling

In this section, we briefly explain how to model DNA by simple graph data structures in our previous work [10]. Remaining parts after this section are based on this graph model.

2.1 Structure

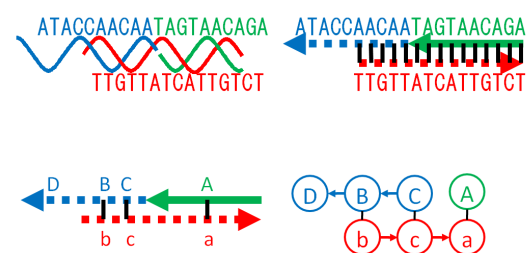


Fig. 1
DNA MODELING

We modeled DNA molecule as a graph data structure to provide a computational model for systems composed of

DNA [10]. For example, Fig. 1 shows the application of the model to a DNA logic gate [3] in a step-by-step manner. Chemically, DNA is a sequence of nucleotides that can be specified by a string of the four elemental bases 'A', 'T', 'G', and 'C'. The duplex structure is formed by hydrogen bonds between complementary base pairs in antiparallel directions (upper left in Fig. 1). Because the target system was a logic gate, we ignored information about the duplex and saved the directions of phosphate backbones by representing a single DNA strand as an arrow and hydrogen bonds as connected lines (upper right in Fig. 1).

We treated a reaction unit of bases as a segment, and a single DNA strand was abstracted into a sequence of segments by allocating a letter to each segment (bottom left in Fig. 1). We used lowercase and uppercase letters to represent information about complementary relationships between segments. For example, 'a' is complementary to 'A'.

Although many kinds of systems are designed using a similar modeling technique, we further abstracted this model as a graph data structure to simplify the reaction rules. We regarded segments as nodes, hydrogen bonds as undirected edges, and phosphate backbones as directed edges (bottom right in Fig. 1). We assumed that one DNA structure corresponds to a connected graph and regarded a disconnected graph as a set of DNA structures.

2.2 Reaction

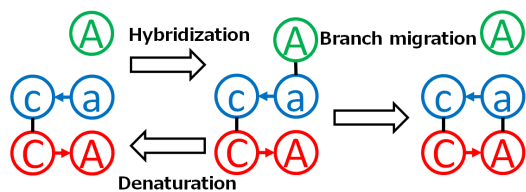


Fig. 2
REACTION RULES

After the DNA graph data structure is thus obtained, we defined three reaction rules, namely, hybridization, denaturation, and branch migration (Fig. 2), because many artificial DNA systems can be developed using only these three simple mechanisms (such as [3], [4], [5], [6]). Hybridization represents a reaction in which antiparallel complementary base pairs bind together with hydrogen bonds. This corresponds to adding a new undirected edge between nodes of uppercase and lowercase letters (transition from the left to center in Fig. 2). Denaturation is the inverse reaction in which hybridized complementary base pairs separate from each other. This corresponds to erasing the undirected edge (transition from the center to left in Fig. 2). Branch migration is a reaction in which an exchange of hydrogen bonds occurs in a single molecule at the branching position of three hybridized strands. This corresponds to transferring an undirected edge (transition from the center to right in Fig. 2).

This data structure and the reaction model are sufficient to represent artificial systems powered by DNA hybridization reactions.

3. Explosion Problem

The combinatorial explosion of molecules is a fundamental problem, especially in simulations of molecular reaction systems including those inside a cell. For example, an unbounded number of structures are produced by a hybridization chain reaction (HCR) that causes a cascade of hybridization reactions triggered by an initiator [6]. In an HCR, there are two hairpin DNA strands at the initial condition of the system and one initiator strand that serves as input (Fig. 3).

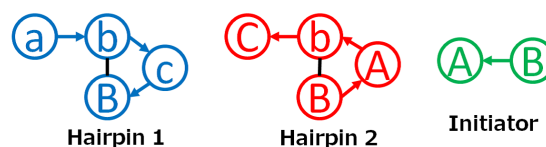


Fig. 3
HCR COMPONENTS IN THE GRAPH MODEL

By adding the initiator to the system, hybridization and branch migration reactions occur alternately and the length of the structure grows unboundedly because of the very large number of copies of hairpin strands (Fig. 4). The figure lists the possible structures of the early stage of HCR using the DNA graph model to illustrate the concept of unbounded growth.

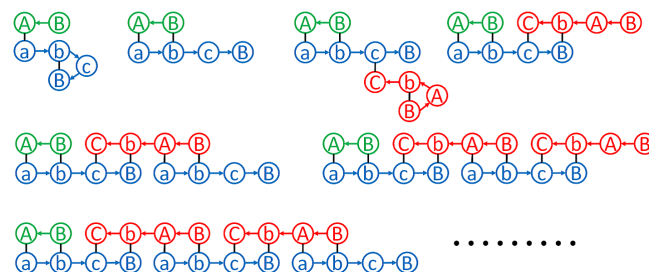


Fig. 4
LIST OF HCR STRUCTURES

Simulating this kind of system is impossible because of the requirement to allocate an unbounded number of variables to each structure.

4. Abstraction by Local Structure

To avoid such unbounded numbers of structures, we introduce an abstraction of the graph model by focusing on the local structure. Although the information about the global structure is lost by the abstraction, using the simulator to design DNA circuits is possible when the outputs are assumed to be single-stranded. The abstraction is done by enumerating possible connecting states of single strands; this is possible

because the number of strands is limited even if the number of structures is unbounded. At least 13 local structures of single strands exist for the HCR reaction, as shown in Fig. 5. Note that each undirected line corresponding to a hydrogen bond contains information about the segment to which it connects but the information is omitted in the figure. By a calculation explained later, the total number of local structures is 126, which means that a finite number is obtained by enumerating local structures.

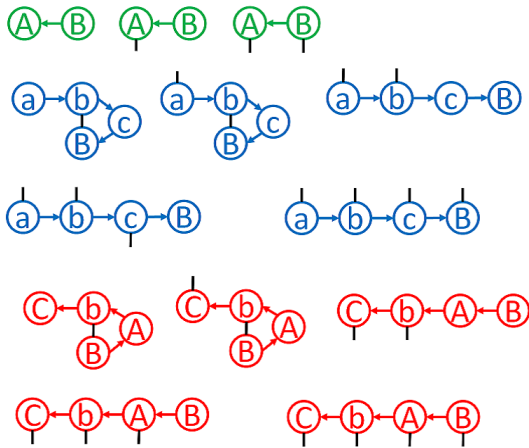


Fig. 5

LOCAL STRUCTURES EXAMPLE

The concept of local structure is defined formally as follows. Assume that an alphabet Σ and a set of single strands $S \subseteq \Sigma^*$ are given in advance, and the binary relation $X \subseteq \Sigma \otimes \Sigma$ is also defined to represent the complementary relationships of segments, where \otimes represents a direct product of sets. By distinguishing all segments of strands, we define the set of local segments $G \subseteq S \otimes \mathbb{N}$ as

$$G = \{(s, i) \mid s \in S, i \in \mathbb{N}, i \leq |s|\},$$

where \mathbb{N} denotes the set of all positive integers, and $|s|$ denotes the length of s . We define a function *LETTER*, which is a map from G to Σ such that for any $g = (s, i) \in G$ and $s = a_1 a_2 a_3 \dots$, $LETTER(g) = a_i$ holds. This function gives the corresponding letter of a given local segment. As a consequence, the set of local structures $L \subseteq S \otimes (G \cup \epsilon)^*$ is defined as

$$L = \{(s, g_1 g_2 \dots g_n) \mid s = a_1 a_2 \dots a_n \in S, \\ \text{either } (a_i, LETTER(g_i)) \in X \text{ or} \\ g_i = \epsilon \text{ holds for all } 1 \leq i \leq n\}.$$

Note that we use ϵ as a symbol to represent unconnected segments, and sequence of ϵ is allowed in $g_1 g_2 \dots$. Thus, $(s, g_1 g_2 \dots g_n)$ corresponds to single-stranded DNA if $g_i = \epsilon$ holds for all $1 \leq i \leq n$.

For example, modeling an HCR by the graph data structure

gives sets

$$\Sigma = \{ 'a', 'A', 'b', 'B', 'c', 'C' \} \\ S = \{ "abcB", "BAbC", "BA" \}$$

and the relation

$$('a', 'A') \in X, ('b', 'B') \in X, \dots$$

Local segments and local structures are defined as

$$G = \{ ("abcB", 1), ("abcB", 2), ("abcB", 3), \\ ("abcB", 4), ("BAbC", 1), ("BAbC", 2), \dots \} \\ L = \{ ("abcB", \epsilon \epsilon \epsilon \epsilon), ("abcB", \epsilon \epsilon \epsilon ("abcB", 2)), \\ ("abcB", \epsilon \epsilon \epsilon ("BAbC", 3)), \dots \}.$$

Enumeration of local structures is performed by finding all possible $l \in L$. To enumerate the total number of local structures from a given alphabet and strands, we define functions *SEGMENTS*, *COMPLEMENTS*, and *CONNECTIONS*. First, *SEGMENTS* is a map from S to 2^G defined as

$$SEGMENTS(s) = \{(s, 1), (s, 2), \dots, (s, |s|)\},$$

which expresses all local segments in a given strand. Next, *COMPLEMENTS* is a map from G to 2^G defined as

$$COMPLEMENTS(g) = \\ \{g' \mid (LETTER(g), LETTER(g')) \in X\}.$$

This finds all segments that are complementary to the given segment. Then, *CONNECTIONS* is a map from S to \mathbb{N} defined as

$$CONNECTIONS(s) = \\ \prod_{g \in SEGMENTS(s)} (|COMPLEMENTS(g)| + 1),$$

where $|COMPLEMENTS(g)|$ denotes the cardinality of set $COMPLEMENTS(g)$. This calculates the number of all combinations of connections from a given strand. Finally, the total number of local structures is calculated by the following expression

$$\sum_{s \in S} CONNECTIONS(s).$$

5. Simulation

This modeling process makes simulation of the concentration changes possible by solving the differential equation using numerical analysis. We defined two kinds of deterministic simulations using either the original or abstracted models of DNA structures. We refer to the simulator based on the original graph and abstracted local model as the original simulator and local simulator, respectively. If a user defines the initial configuration as a set of structures and their concentrations, the simulators return the calculation results and the user can trace the concentration changes. These simulators perform the calculations in two stages: enumerating structures that can be constructed from initial structures, and analyzing the concentration changes numerically.

At the beginning of a simulation, the simulators enumerate whole structures in a system to determine the number of variables and their relationships, where each variable represents the concentration of the corresponding structure. The total number of structures is determined by applying the three reaction rules to the initial set of structures as shown in the HCR reaction example in the previous section.

The original simulator enumerates possible graph structures with two restrictions. First, a structure cannot contain two or more identical single strands; this prevents the combinatorial explosion of structures that contain a repeated sub-structure. Second, structures that have a concentration less than 10^{-5} are disregarded to ignore unimportant structures that may not be the main products of a simulation. To implement this feature, the original simulator generates structures dynamically and checks whether the concentration of each structure exceeds the threshold given in advance. More concretely, the period of a simulation is divided into intervals, and the simulator checks the concentration at the beginning of each interval. The simulator then continues the rest of the simulation with the remaining structures whose concentration does not exceed the threshold.

On the other hand, the local simulator enumerates all of the possible local structures without restriction as explained in section 4.

After enumerating structures and reactions among them, the simulators assign variables to each structure and define differential equations using chemical kinetics. The simulators formalize all three reactions. Figs. 6 and 7 show schematic examples of the original and local simulators, respectively. According to the reactions shown in the figures, differential equations for the original simulation are

$$\begin{aligned} \frac{d}{dt}C_1 &= -k_d C_1 \\ \frac{d}{dt}C_2 &= k_d C_1 \\ \frac{d}{dt}C_3 &= k_d C_1, \end{aligned}$$

and differential equations for the local simulation are

$$\begin{aligned} \frac{d}{dt}C_4 &= -k_d R(C_4 C_5) \\ \frac{d}{dt}C_5 &= -k_d R(C_4 C_5) \\ \frac{d}{dt}C_6 &= k_d R(C_4 C_5) \\ \frac{d}{dt}C_7 &= k_d R(C_4 C_5) \end{aligned}$$

where k_d is the reaction rates for denaturation and C_1, \dots, C_7 are the variables assigned to each structure as a concentration. Because many reactions occur in a single simulation, each of $\frac{d}{dt}C_1, \dots$ is defined by summing up all of the reactions on which the corresponding structure depends. In the local simulation, we introduce an arrangement (represented by the symbol R) in the differential equations compared with ordinary chemical kinetics. This R is introduced to emulate multi-

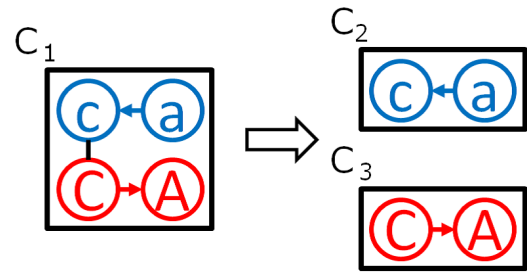


Fig. 6

FORMALIZATION FOR DENATURATION IN ORIGINAL SIMULATION

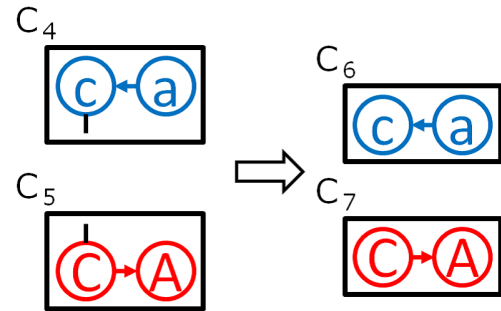


Fig. 7

FORMALIZATION FOR DENATURATION IN LOCAL SIMULATION

molecular reactions as unimolecular reactions because denaturation (especially denaturation reactions that separate multiple segments in a row) and branch migration must be unimolecular reactions. This calculates the ratio of concentration among all possible connections from reacting segments. Suppose that C_l denotes the concentration of the local structure l , and function $CONNECTED$ is a map from G to 2^L as

$$\begin{aligned} CONNECTED(g) = \\ \{l \mid s \in S, \vec{g} \in (G \cup \epsilon)^*, l = (s, \vec{g}) \in L, g \text{ appears in } \vec{g}\}, \end{aligned}$$

where “ g appears in \vec{g} ” means that $\vec{g} = g_1 g_2 \dots g_n$ and $g = g_i$ holds for some i . $CONNECTED$ finds all local structures that are connected to the given local segment. $R(C_{l_1} C_{l_2})$ for denaturation between the segments g_1 of local structure l_1 and g_2 of l_2 is defined as

$$\frac{C_{l_1} C_{l_2}}{\sum_{l \in CONNECTED(g_1)} C_l},$$

which is equivalent to

$$\frac{C_{l_1} C_{l_2}}{\sum_{l \in CONNECTED(g_2)} C_l}.$$

The rate of each reaction is defined by rule of thumb, and the kinetics of hybridization and branch migration are fixed. Only the kinetic velocity of denaturation is calculated according to the information of segments that are separating. We use a Runge-Kutta-Fehlberg-4,5 method [13] to analyze the

differential equations; this is a numerical analysis technique using step size controls.

6. Enumeration and Simulation Results

We have described two types of simulations for DNA reaction systems. Experiments were conducted to examine and compare their features. As an environment for the experiments, we used a computer with Intel Core2 Duo (2.66GHz) CPU, 4026MB RAM, Windows Vista OS, and executed the simulator on Java 1.6.

6.1 Enumeration Efficiency

The efficiency of the two types of simulations was tested in terms of the number of structures. As a benchmark, we generated a random system as a random sequence of letters, which determines the set S . Note that the size of set Σ was fixed to 14 for all simulations. Such random systems were actually generated in our previous work for the automatic design of DNA logic gates. We first fixed a maximum size of S , and then we generated and simulated 200 random systems to obtain the average and maximum numbers of structures. After that, we took another maximum size in turn and repeated the calculation for each maximum size. We tried 21 different sizes (whose values were suitable for simulations), which range over the x -axis of Figs. 8 and 9.

The figures show the average and maximum number of structures produced by four types of simulation: original simulation without threshold, local simulation, original simulation with threshold, and stochastic simulation. Note that original simulations with or without threshold impose the restriction on DNA structures mentioned above. The x -axis of the figure corresponds to the maximum number of local segments, which is the number of letters in S to generate a random system. The y -axis of the figure corresponds to the number of structures for each simulation.

The stochastic simulation was not explained above because it is not an integral part of this research. Stochastic simulation is an algorithm for simulating discrete chemical reaction systems using a statistical simulation method. This statistical method simulates chemical reactions stochastically one by one according to the distribution of possibility of each reaction. We actually implemented Gillespie's algorithm [14] for this method.

Note that the results of original simulation with threshold and stochastic simulation are only shown as references in the figures. Direct comparison of the results is not fair because the values for the original simulation with threshold and the stochastic simulation depend on parameters such as threshold and copy number.

As expected, the original simulation without threshold exhibited faster combinatorial explosion than the others because entire combinations of structures were tested by the execution. Completion of the original simulation with a size greater than 40 was impossible due to an out-of-memory error. The increase in the number of structures in the local simulation seemed to

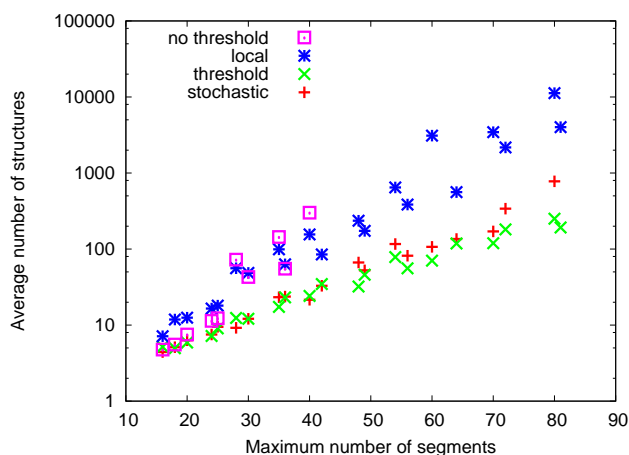


Fig. 8

AVERAGE NUMBER OF STRUCTURES

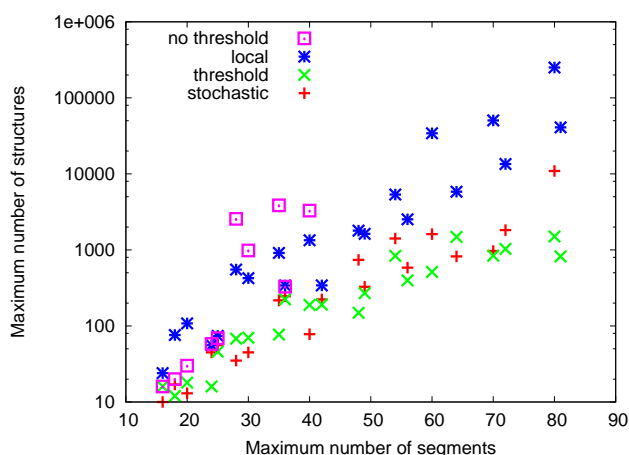


Fig. 9

MAXIMUM NUMBER OF STRUCTURES

be slower than that of the original simulation without threshold because of the limit on the number of local structures.

These results indicate that an appropriate model and simulation are necessary for the efficient enumeration and simulation of DNA hybridization systems.

6.2 Simulation of Entropy-driven Gate

We choose entropy-driven amplifier gate [5] as a benchmark to verify that the two simulations can predict the behavior of the DNA gate correctly. This gate is composed of a three-strand structure with one fuel strand, and an output strand is emitted using an input strand as a catalyst.

Figs. 10 and 11 show the results of the original and local simulations, respectively. The x - and y -axes in the figures correspond to the virtual time of the simulation and the virtual concentration of the output strand, respectively. Changes in the first 2000 time units were due to constructing the gate, and the input was added after 2000 time units had elapsed. The

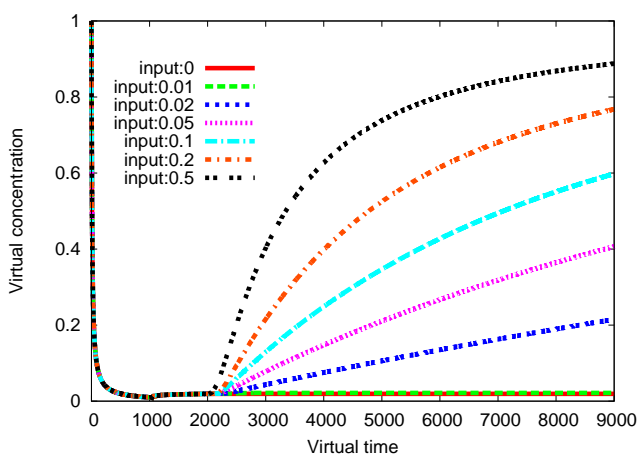


Fig. 10
ORIGINAL SIMULATION

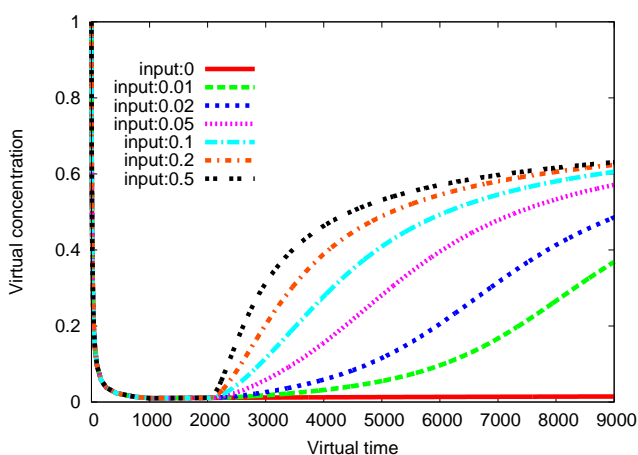


Fig. 11
LOCAL SIMULATION

graph legends indicate the concentrations of inputs.

Because most of the results exhibit catalytic activity in which the concentration of the output increases more than that of the input, it is possible to ensure that both simulations predict the behavior correctly. However, in the original simulation where 0.01 concentration of input was added, the output concentration did not increase because the input concentration was too small to exceed the threshold. In contrast, the output concentration did increase for the local simulation, which is a more precise simulation of the system than the original simulation.

7. Discussion

A limit to designing very complex DNA systems lies in the combinatorial explosion problem of the DNA structures. This is critical because preventing the combinatorial explosion by enumerating all of the possible structures is difficult where an unbounded number of structures can occur. Imposing a

threshold or artificial limitations on the model of structures did not eliminate the problem and introduced the possibility of incorrect simulation, as the results in the previous section showed.

A new approach to avoiding the combinatorial explosion was proposed that focused on the local structure, and the efficiency of this approach was better than the original model. The rapid increase in the number of structures was reduced in the local simulation. Considering all of the possible local structures in a simulation was possible because none of the structures were ignored as the result of imposing an artificial threshold. The strongest aspect of the model was the ability to express any kind of structure at the expense of losing some part of the information, even in the case where unbounded structures were involved.

The HCR [6] explained in Section 3 is a typical system that produces an unbounded number of structures, and requires infinite-dimensional ordinary differential equations (ODEs) to be solved to compute concentrations of all possible structures. The original simulator ignores structures with a small concentration and approximates infinite-dimensional ODEs by finite-dimensional ODEs. On the other hand, the local simulator focuses on local structures and obtains finite-dimensional ODEs. The resulting ODEs are not an approximation but exactly model the behavior of local structures.

A question that naturally arises here is whether it is possible to obtain concentrations of global structures from the distributed concentrations of local structures. This seems possible if we make “maximum entropy assumption” as in the related work [11], but it is not always the case that the assumption is true. For example, in the case of the HCR, the maximum entropy assumption induces a Poisson distribution on the length of structures, and the local simulator gives the result in Fig. 13, while the original simulator gives the result in Fig. 12. The x - and y -axes in the figures correspond to the virtual time of the simulation and the virtual concentration of the DNA structures, respectively. The graph legends indicate the number of hairpin strands that connect to the initiator in a structure. In this comparison, the original simulator is considered more accurate because it enumerates an enough number of structures. The results indicate that the local simulator does not recover the concentrations of global structures accurately.

While the purpose of the related work [11] was to theoretically compute equilibrium state of hybridization reaction system based on locality, we gave a concrete simulator in this work that can trace the time change of concentration. Though this work shares the basic idea with the related work [12], we defined the local structure for our original purpose, which is to simulate DNA hybridization systems. We actually showed that the local simulation was effective in enumeration of structures and more precise than the original simulation with threshold. Because the targets of our automatic design were gates that output single-stranded DNA, the modeling using the local structure can be regarded as a novel abstraction that serves our purpose.

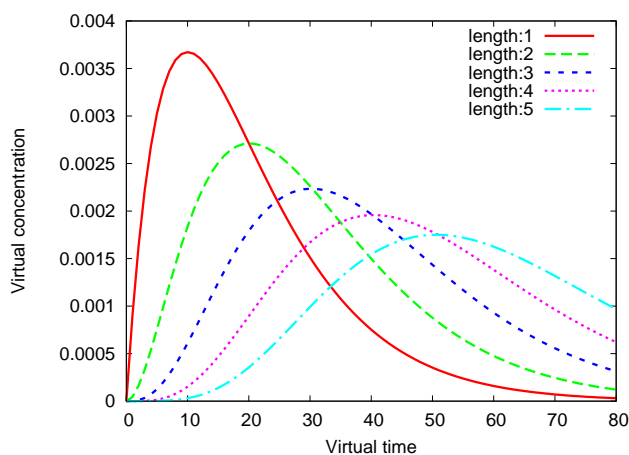


Fig. 12

ORIGINAL SIMULATION OF HCR

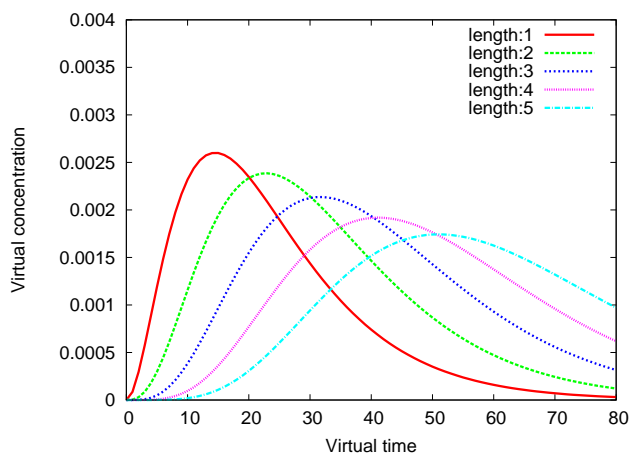


Fig. 13

GLOBAL INFORMATION RECOVERED BY LOCAL SIMULATION OF HCR

8. Conclusion

DNA hybridization systems have been applied to a wide range of applications including molecular robotics, nano-scale structures, and medication control. Because selecting combinations of molecules to achieve some desired functionality is difficult for humans, we previously proposed an automatic design method using an evolutionary computation. Modeling for molecules and reactions was defined by regarding molecules as a graph data structure. Because the automatic design method required efficient enumeration and simulation to avoid combinatorial explosion, we abstracted the model to limit the number of structures by enumeration, even if an unbounded number of structures can be constructed. On the basis of this modeling technique, we developed a simulator and investigated the efficiency in the enumeration of structures and the prediction of system behavior. Synthesis of large systems that are more complex than human beings can design will be possible using this new abstracted model.

References

- [1] Rothemund, P. W. K.: Folding DNA to create nanoscale shapes and patterns, *Nature*, Vol. 440, No. 7082, pp. 297–302 (2006).
- [2] Andersen, E. S., Dong, M., Nielsen, M. M., Jahn, K., Subramani, R., Mamdouh, W., Golas, M. M., Sander, B., Stark, H., Oliveira, C. L. P., Pedersen, J. S., Birkedal, V., Besenbacher, F., Gothelf, K. V. and Kjems, J.: Self-assembly of a nanoscale DNA box with a controllable lid, *Nature*, Vol. 459, No. 7243, pp. 73–76 (2009).
- [3] Seelig, G., Soloveichik, D., Zhang, D. Y. and Winfree, E.: Enzyme-Free Nucleic Acid Logic Circuits, *Science*, Vol. 314, No. 5805, pp. 1585–1588 (2006).
- [4] Qian, L. and Winfree, E.: A simple DNA gate motif for synthesizing large-scale circuits, *DNA Computing*, Vol. 5347 of LNCS, pp. 70–89 (2008).
- [5] Zhang, D. Y., Turberfield, A. J., Yurke, B. and Winfree, E.: Engineering Entropy-Driven Reactions and Networks Catalyzed by DNA, *Science*, Vol. 318, No. 5853, pp. 1121–1125 (2007).
- [6] Dirks, R. M. and Pierce, N. A.: Triggered amplification by hybridization chain reaction, *Proc. Natl. Acad. Sci. U. S. A.*, Vol. 101, No. 43, pp. 15275–15278 (2004).
- [7] Yin, P., Choi, H. M. T., Calvert, C. R. and Pierce, N. A.: Programming biomolecular self-assembly pathways, *Nature*, Vol. 451, No. 7176, pp. 318–322 (2008).
- [8] Phillips, A. and Cardelli, L.: A programming language for composable DNA circuits, *J. R. Soc. Interface*, Vol. 6, pp. 419–436 (2009).
- [9] Douglas, S. M., Marblestone, A. H., Teerapittayanon, S., Vazquez, A., Church, G. M. and Shih, W. M.: Rapid prototyping of 3D DNA-origami shapes with caDNAno, *Nucleic Acids Res.*, Vol. 37, No. 15, pp. 5001–5006 (2009).
- [10] Kawamata, I., Tanaka, F. and Hagiya, M.: Automatic Design of DNA Logic Gates Based on Kinetic Simulation, *DNA Computing and Molecular Programming*, Vol. 5877 of LNCS, pp. 88–96 (2009).
- [11] Kobayashi, S.: A New Approach to Computing Equilibrium State of Combinatorial Hybridization Reaction Systems, in *Bio-Inspired Models of Network, Information and Computing Systems*, pp. 330–335 IEEE (2008).
- [12] Danos, V., Feret, J., Fontana, W. and Krivine, J.: Abstract Interpretation of Cellular Signalling Networks, in *Verification, Model Checking, and Abstract Interpretation*, pp. 83–97 Springer (2008).
- [13] Fehlberg, E.: Klassische Runge-Kutta-Formeln vierter und niedrigerer Ordnung mit Schrittweiten-Kontrolle und ihre Anwendung auf W"armerleitungsprobleme, *Computing*, Vol. 6, No. 1, pp. 61–71 (1970).
- [14] Gillespie, D.: Exact stochastic simulation of coupled chemical reactions, *The journal of physical chemistry*, Vol. 81, No. 25, pp. 2340–2361 (1977).

A Heuristic Line Balancing Algorithm Accounting for Component Mounting Order

Hiroshige Tozaki, Hidenori Ohta and Mario Nakamori

Department of Computer Science, Tokyo University of Agriculture and Technology,
Koganei, Tokyo, Japan

Abstract - A printed circuit board production line is composed of several component-mounting machines arranged in series that mount components onto each board. Since the performance of the line is determined by the most time consuming machine, the line balancing problem occurs that makes the mounting time by machines as equal as possible. Conventional line balancing procedures use time estimates based on the total number of components to be mounted under each machine. The actual mounting time, however, is often quite different from such estimates, and a substantial discrepancy arises between the actual production time and the estimated one of the line. In the present paper, a new estimation method of mounting time is proposed based on the calculation of the length of mounting paths, and also a heuristic algorithm of the line balancing problem is proposed. The proposed algorithm is shown by computer experiments to provide better results than conventional procedures.

Keywords: line balancing, component mounting

1 Introduction

The process of mounting electronic components, such as integrated circuits, resistors, and condensers, onto printed circuit boards (PCB) is generally a bottleneck in the production of such boards and is the dominant process affecting production efficiency. This process is carried out on a line formed of several component mounting machines ("machines" in abbreviation) connected in series. Previous studies aiming at reducing production time are either on rough optimization of the line or on individual optimization of each machine movement [1], and studies on the total optimization including both line and individual machine are scarce.

The present paper addresses the problem of allocating components in a component mounting line composed of machines. The target problem of allocating components in a component mounting line is to assign multiple components to each machine such that the production efficiency is the maximum. Both this problem and that of determining the component mounting order in each machine have influence to each other. As a result, even if each is optimized independently, the results will not necessarily result in overall optimization when combined. The allocation of components to each machine, however, has been optimized

without taking the issue of component mounting order into consideration. Furthermore, estimation of the mounting time has been based on the component numbers of the components to be allocated by each machine [2], and no procedure accounting for the mounting path in each machine has been considered. Estimation methods based on the number of components show large discrepancies with actual production time. In the present paper we propose a heuristic algorithm that allocates components while it calculates the mounting paths in each machine.

2 Formulation of the Problem

Each component mounting machine contains a head equipped with multiple vacuum nozzles that pick components and mount them on the board. The action of a machine is as follows: Using its vacuum nozzles, the head first picks up multiple components that have been positioned on the supply feeder, transports them to the location on the board where one of the components is to be mounted, mounts one component, then moves to the location where a second component is needed and mounts that component and so on, repeating until all of the components it picked up have been mounted, after which the head returns to the supply feeder to pick up another batch of components. The sequence of actions from the pick-up to the return of the head to the supply tray is called a turn.

The present paper addresses the component allocation problem while at the same time taking into consideration the mounting paths of each machine. A line is assessed based on the time spent per single board by the bottleneck machine. This time is strongly affected by the travel distance of the head, which is defined as the Chebyshev distance because head is driven by motors that operate independently in the x and y directions. Therefore, our algorithm calculates the head travel distance for each machine considering both component allocation and component mounting order, evaluating the component allocation by the maximum head travel distance.

Since this paper focuses on component allocation and component mounting order, the problem of component pick-up is only briefly addressed; all components are assumed to be picked up at an identical point on the supply feeder; the distances between nozzles are also neglected by assuming that the head makes no extra motions to pick up components. For the sake of simplicity, it is also assumed that all the

components could be picked up by any nozzle, and that there is no upper limit on the number of component types that could be placed on the supply feeder.

The above problem is formulated as an integer programming problem as follows:

$$\text{minimize } z = \max(T_j) \quad j = 1, \dots, jsize \quad (1)$$

subject to

$$T_j = \sum_{r=1}^{m_j} \left\{ t(c(0), c(m_{jr})) + \sum_{i=1}^{m_r} t(c(i-1), c(i)) \right\} \quad (2)$$

$$m_{jr} \leq N \quad (3)$$

$$t(a, b) = \max(|x_b - x_a|, |y_b - y_a|) \quad (4)$$

$$M = \sum_{j=1}^{jsize} \sum_{r=1}^{m_j} m_{jr} \quad (5)$$

z : objective function

T_j : total travel distance of machine j

$jsize$: total number of machines composing the production line

m_j : number of component mounting paths in machine j

$c(i)$: coordinates of i th component on the mounting path

$c(0)$: coordinates of the supplyfeeder

m_{jr} : total number of components included in path r of machine j

N : number of nozzles

x_a : x coordinate of component a

y_a : y coordinate of component a

M : total number of components to be mounted

Formula (1) is the objective function and states that the maximum value of the total head travel distance of each machine calculated in (2) should be minimized. Constraint (3) limits the maximum number of components that can be mounted in one turn. Constraint (4) defines the Chebyshev distance between two component insertion points. Constraint (5) expresses the number of components mounted on a single board.

3 Conventional procedures

In conventional procedures, once the components have been allocated to machines in a balanced manner according to the number of components, the mounting order of the components in each machine is determined and the head path is created. This procedure is easily adaptable to simulated annealing as follows: we use the allocation obtained from the above conventional method as the initial solution; in order to improve temporary solution, we exchange the components allocated to the bottleneck machine with those allocated to other machines or pass the components allocated to the bottleneck machine to another machine. We call this method "extended conventional procedure." Figure 1 shows a flow diagram for the extended conventional procedure.

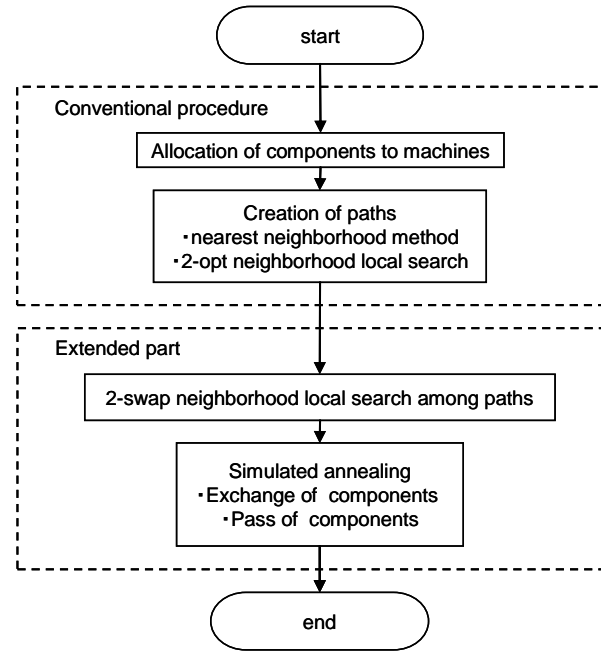


Figure 1 Extended conventional procedure

4 Proposed algorithm

In the present paper we propose a hybrid genetic algorithm, which is a combined version of a genetic algorithm with local search technique. Figure 2 shows a flow diagram of the proposed algorithm.

Incorporating local search technique into the genetic algorithm strengthens weak points of genetic algorithm and provides a more precise solution. In the algorithm proposed here, local search is used for initial solution generation, mutation and offspring solution improvement.

4.1 Expression of solutions

In the proposed algorithm, all machines are assumed to mount their components in the same number of turns. When the number of turns increases, the head must move between the supply tray and the board more frequently, so it is desirable that the number of turns for each machine be reduced as much as possible. The minimum number of turns necessary for a single machine is obtained by (6). Note that, depending on the number of machines and components available for mounting, there will be turns during which no components are picked up by some of the nozzles.

$$m_j = \left\lceil \frac{M}{N \times jsize} \right\rceil \quad (6)$$

The components are assigned individual numbers, and the solution is expressed by a row of these component numbers aligned as shown in Fig. 3. If the arranged components are separated into blocks based on a fixed number of components starting from the first, this will correspond with

their allocation to the machines and paths. Furthermore, the sequence of component numbers within a turn will reflect the mounting order. Figure 3 shows an example involving two machines and six nozzles, where each machine carries out two turns of component mounting. Unused nozzles are flagged with the letter “e” in place of the component number. Figure 4 shows the component allocation corresponding to the solution shown in Figure 3 and the paths of the head.

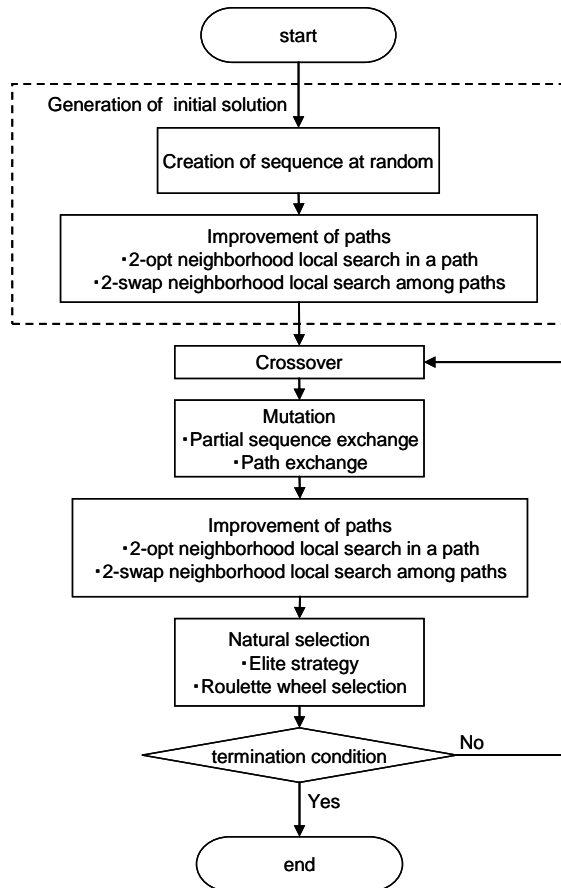


Figure 2 Proposed algorithm

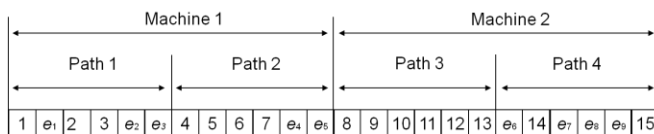


Figure 3 Expression of the solution

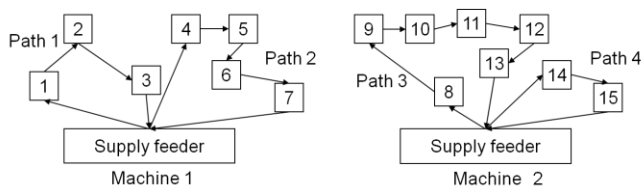


Figure 4 Component allocation corresponding to Fig 3

4.1.1 Fitness of solution

The evaluation value is the head travel distance of the bottleneck machine. Accordingly, when we select parents and individuals during crossover based on the score, we are not considering any machines other than the one that has become the bottleneck. However, the path lengths in the other machines might have a significant influence on the solution generated by crossover and mutation. Therefore, in this paper, in addition to the interconnect length in the machine that is identified as the bottleneck, we calculate the fitness value of the solution while incorporating the total interconnect length for all machine heads. The fitness value of each individual is the inverse of the square of the sum of the head travel distance of the bottleneck machine and the head travel distances of all the other machines. Equation (7) shows how to calculate the total path length L and Equation (8) shows how to calculate the fitness value f , where S is the size of the area.

$$L = \sum_{j=1}^{jsize} T_j \tag{7}$$

$$f = \left(\frac{1}{z + L} \right)^2 \times S \tag{8}$$

4.2 Generation of the initial solution

Our genetic algorithm requires initial solutions as many as the pre-determined number of individuals. Each of the initial solutions is generated using the following two steps:

- (1) A sequence is created at random.
- (2) Sequences are improved by 2-opt and 2-swap neighborhood local search.

4.3 Crossover

Crossover is performed in the proposed algorithm in the order from (1) to (3) below, to obtain a child:

- (1) A partial sequence of random length is selected from parent 1.
- (2) A sequence is formed by removing all of the elements included in the partial sequence selected in step (1) from parent 2.
- (3) The partial sequence selected in step (1) from parent 1 is inserted into the sequence created in step (2) from parent 2. The insertion location is the same location at which the selected sequence had previously existed in parent 1.

Figure 5 shows an example of crossover. For two parent individuals, crossovers are performed that are randomly selected at a probability proportional to their fitness values. The above steps (1)-(3) are executed for parents 1 and parent 2 as well as for parent 2 and parent 1, so we have two children from one couple.

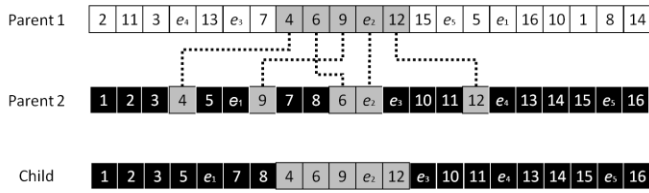


Figure 5 Crossover

4.4 Mutation

Two types of mutations are proposed here, partial sequence exchange and path exchange. The parent individuals used to generate the mutation are selected at random from all the individuals.

4.4.1 Mutation by partial sequence exchange

In mutation by partial sequence exchange, two partial sequences are selected at random from each selected individual and exchanged. The length of the partial sequence is changed at random within a specified range. Figure 6 shows a typical exchange when the partial sequence length is two.

4.4.2 Mutation by partial sequence exchange

In mutation by path exchange, two partial sequences corresponding to paths are selected at random from each selected individual and exchanged. Figure 7 shows a typical mutation by path exchange.

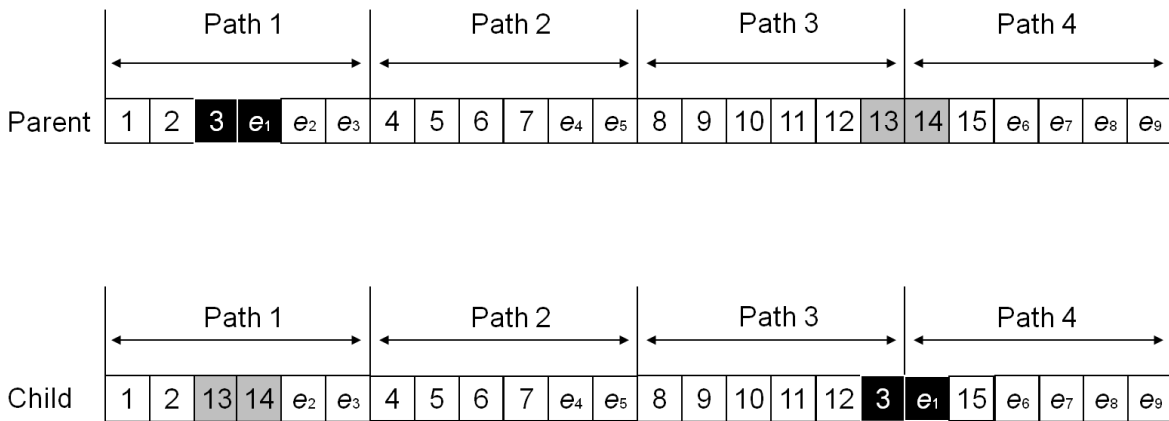


Figure 6 Partial sequence exchange

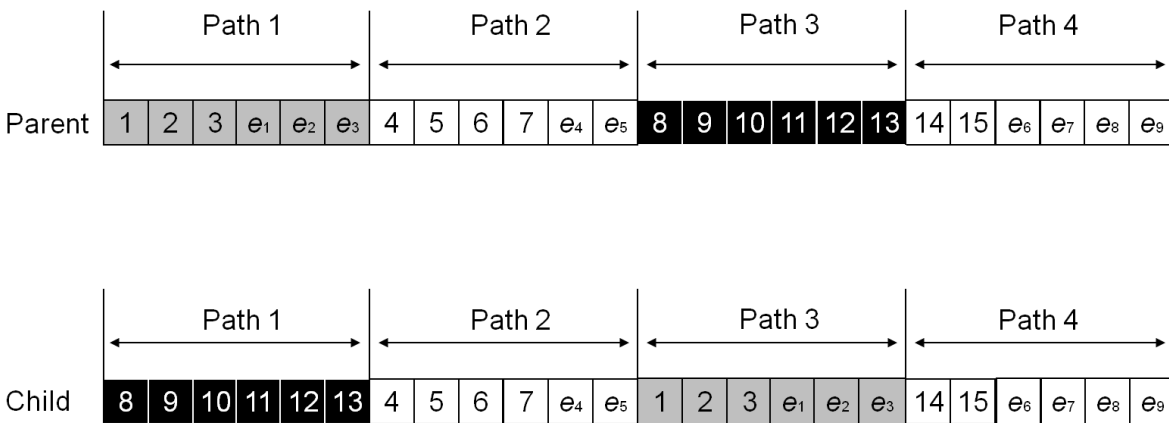


Figure 7 Path exchange

4.5 Natural selection

30% of the present generation are selected by the elite strategy as individuals left to the next generation. Some of the remaining 70% unselected by the elite strategy are decided by roulette wheel selection to be individuals left to the next generation. In roulette wheel selection, individuals are chosen at a probability proportional to fitness.

5 Computer experiment

We made a comparative experiment on a computer to evaluate the performance of the proposed procedure. The computer environment is an Intel Core2 Duo 3.00 GHz CPU with 1.96 GB of memory. The programming language used was C++.

The supposed line contained six machines, each with a head equipped with 12 nozzles. The starting point for insertion was at a location 350 mm in the y direction from the center of gravity of the board. Nine sets of data were prepared for entry, which randomly specified the coordinates of the mounting locations on a board 100 mm wide by 100 mm deep in size.

The proposed algorithm used 25 individuals and the ending condition was the 3000th generation. The values for other parameters were specified appropriately based on a preliminary experiment. Figure 8 shows the relation between the scores for the proposed procedure and the extended conventional procedure as well as the calculation time and shows that the proposed procedure

quickly obtains better solutions than the extended conventional one.

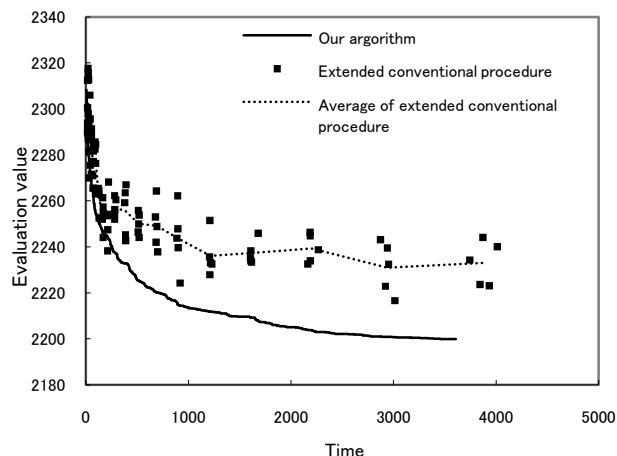


Figure 8 Evaluation value and time

Table 1 presents the evaluation values for the proposed procedure, the conventional procedure, and the extended conventional procedure obtained after a sufficient length of time, for each data set entered. The proposed procedure obtained the best evaluation values in all of the nine data sets of various types. The extended conventional procedure showed a mean improvement over the conventional procedure of 13.9%, while the proposed procedure showed a mean improvement over the conventional procedure of 14.6%.

Table 1 Comparison of conventional procedure, extended conventional procedure and our algorithm

Number of mounting points	Conventional procedure		Extended conventional procedure		Our algorithm		
	Search time (sec)	Evaluation value	Search time (sec)	Evaluation value	Search time (sec)	Evaluation value	
1	100	0.005	1694.93	837.08	1420.66	495.92	1414.69
2	100	-	1674.24	840.00	1405.06	525.12	1404.95
3	100	-	1675.75	829.32	1441.25	520.31	1423.67
4	200	0.005	2594.38	3896.77	2232.47	3632.48	2197.78
5	200	0.016	2594.04	3922.46	2225.78	3648.23	2194.31
6	200	-	2585.95	3880.22	2210.18	3869.80	2181.23
7	400	0.130	5048.42	42167.83	4381.65	34179.47	4357.23
8	400	0.125	5041.57	39346.17	4381.29	31826.50	4360.36
9	400	0.130	5052.73	38433.20	4373.88	31012.80	4342.90

6 Conclusions

In this paper we proposed a procedure for allocating components to machines on a printed circuit board production line by actually creating component mounting paths, rather than by creating rough estimates using only the number of components to be mounted. Computer experiment shows that the proposed procedure provides good solutions than the conventional procedure or an extended version of the conventional procedure. Issues to be addressed in the future include incorporating limitations specific to printed circuit board production lines, such as number of component types and component heights.

7 References

- [1] Masri Ayob, Graham Kendall. A survey of surface mount device placement machine optimisation: Machine classification. *European Journal of Operational Research* 186 (2008) , 893–914
- [2] Osman Kulak, Ihsan Onur Yilmaz. Hans-Otto Günther, A GA-based solution approach for balancing printed circuit board assembly lines. *OR Spectrum* 30 (2008), 469–491
- [3] Sun, D.S., Lee, T.E., Kim, K.H.: Component Allocation and Feeder Arrangement for a Dual-Gantry Multi-Head Surface Mounting Placement Tool. *International Journal of Production Economics*, Vol.95, pp.245?264 (2005).
- [4] Yamada, T., Miyashiro, R., and Nakamori, M.: An Algorithm of Feeder Arrangement and Pick up Sequencing of Component Placement Machine on Printed Circuit Board, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, pp.403-409 (2005).
- [5] Ahmadi, R. H. and Mamer, J. W.: Routing Heuristics for Automated Pick and Place Machines, *European Journal of Operational Research*, Vol. 117, pp. 533?552 (1999).
- [6] Burke, E. K., Cowling, P. I. and Keuthen, R.: New Models and Heuristics for Component Placement in Printed Circuit Board Assembly, *International Conference on Information Intelligence and Systems*, pp. 133?140 (1999).
- [7] Burke, E. K., Cowling, P. I. and Keuthen, R.: Effective Heuristic and Metaheuristic Approaches to Optimize Component Placement in Printed Circuit Board Assembly, *Evolutionary Computation 2000 Proceedings of the 2000 Congress*, pp. 301?308 (2000).
- [8] Burke, E. K., Cowling, P. I. and Keuthen, R.: The Printed Circuit Board Assembly Problem : Heuristic Approaches for Multi-Headed Placement Machinery, *Proceedings of the International Conference on Artificial Intelligence ICAI'2001* , pp. 1456?1462 (2001).
- [9] Hackman, S. T., Magazine, M. J. and Wee, T. S.: Fast, Effective Algorithms for Simple Assembly Line Balancing Problems, *Operations Research*, Vol. 37, pp. 916?924 (1989).

SESSION

NOVEL APPLICATIONS AND ALGORITHMS + CUDA + GPU + GPGPU + MULTI-CORE + CLUSTER COMPUTING + I/O SYSTEMS + TOOLS

Chair(s)

Prof. Hamid R. Arabnia

Scalable Data-Privatization Threading for Hybrid MPI/OpenMP Parallelization of Molecular Dynamics

Manaschai Kunaseth¹, David F. Richards², James N. Glosli²,
Rajiv K. Kalia¹, Aiichiro Nakano¹, Priya Vashishta¹

¹Departments of Computer Science, Physics, Material Science,
University of Southern California, Los Angeles, CA 90089-0242, USA
{kunaseth, rkalia, anakano, priyav}@usc.edu

²Lawrence Livermore National Laboratory, Livermore, CA 94550, USA
{richards12, glosli}@llnl.gov

Abstract— Calculation of the Coulomb potential in the molecular dynamics code ddcMD has been parallelized based on a hybrid MPI/OpenMP scheme. The explicit pair kernel of the particle-particle/particle-mesh algorithm is multi-threaded using OpenMP, while communication between multicore nodes is handled by MPI. We have designed a load balancing spanning forest (LBSF) partitioning algorithm, which combines: 1) fine-grain dynamic load balancing; and 2) minimal memory-footprint data privatization via nucleation-growth allocation. This algorithm reduces the memory requirement for thread-private data from $O(np)$ to $O(n + p^{1/3}n^{2/3})$ —amounting to 75% memory saving for $p = 16$ threads working on $n = 8,192$ particles, while maintaining the average thread-level load-imbalance less than 5%. Strong-scaling speedup for the kernel is 14.4 with 16-way threading on a four quad-core AMD Opteron node. In addition, our MPI/OpenMP code shows 2.58 \times and 2.16 \times speedups over the MPI-only implementation, respectively, for 0.84 and 1.68 million particles systems on 32,768 cores of BlueGene/P.

Keywords: Hybrid MPI/OpenMP Parallelization; Thread Scheduling; Memory Optimization; Load Balancing; Parallel Molecular Dynamics

I. INTRODUCTION

Molecular dynamics (MD) simulation is widely used to study material properties at the atomistic level. Large-scale MD simulations are beginning to address broad problems [1-6], but increasingly large computing power is needed to encompass even larger spatiotemporal scales. For example, Glosli *et al.* performed a massively parallel MD simulation involving 62 billion particles using the MD code ddcMD, which demonstrated excellent performance and scalability [7].

Due to shifting trends in computer architecture, improvements in computing power are now gained using multicore architectures instead of increased clock speed. Furthermore, the number of cores per chip is expected to continue to grow. As a consequence, the performance of traditional parallel applications, which are solely based on the message passing interface (MPI), is expected to degrade

substantially [8]. Hierarchical parallelization frameworks, which integrate several parallel methods to provide different levels of parallelism, have been proposed as a solution to this scalability problem on multicore platforms [6, 9-11].

Hybrid parallelization based on MPI/threading schemes will likely replace MPI-only parallel MD. However, efficiently integrating a multi-threading framework into an existing MPI-only code is difficult for several reasons: 1) highly overlapped memory layout in typical MD codes incurs serious race condition; 2) naïve threading algorithms usually create significant overhead, and limit the threading speedup for a large number of threads; and 3) dynamic nature of MD requires low-overhead dynamic load balancing for threads to maintain good performance [12].

To address these issues, we have designed a load balancing spanning forest (LBSF) partitioning algorithm, which combines: 1) fine-grain dynamic load balancing; and 2) minimal memory-footprint data privatization via nucleation-growth allocation. We have implemented this algorithm in ddcMD and demonstrated that the hybrid MPI/threading scheme outperforms MPI-only scheme in terms of the strong scaling of large-scale problems.

This paper is organized as follows. Section II summarizes the hierarchy of parallel operations in ddcMD, followed by the description and analysis of the proposed data-privatization algorithm in section III. Section IV evaluates the performance of the hybrid parallelization algorithm, and conclusions are drawn in section V.

II. DOMAIN DECOMPOSITION MOLECULAR DYNAMICS

Molecular dynamics simulation follows the phase-space trajectories of an N -particle system, where the forces between particles are given by the gradient of a potential energy function $\phi(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N)$. Positions and velocities of all particles are updated at each MD step by numerically integrating coupled ordinary differential equations. The dominant computation of MD simulations is the evaluation of

the potential energy function and associated forces. One model of great physical importance is the interaction between a collection of point charges, which is described by the long-range, pair-wise Coulomb field $1/r$ (r is the interparticle distance), requiring $O(N^2)$ operations to evaluate. Many methods exist to reduce this computational complexity [13–15]. We focus on the highly efficient particle-particle/particle-mesh (PPPM) method [13]. In PPPM the Coulomb potential is decomposed into two parts: A short-range part that converges quickly in real space and a long-range part that converges quickly in reciprocal space. The split of work between the short-range and long-range part is controlled through a “screening parameter” α . With the appropriate choice of α , computational complexity for these methods can be reduced to $O(N \log N)$.

Because the long-range part of the Coulomb potential can be threaded easily (as a parallel loop over many individual 1D fast Fourier transforms), this paper explores efficient parallelization of the more challenging short-range part of the Coulomb potential using OpenMP threading. The short-range part is a sum over pairs: $\phi = \sum_{i < j} q_i q_j \text{erfc}(\alpha r_{ij}) / r_{ij}$, where q_i is the charge of particle i and r_{ij} is the separation between particles i and j . Though this work is focused on this particular pair function, much of the work can be readily applied to other pair functions. In addition to this intranode parallelization, the ddcMD code is already parallelized across nodes using a particle-based domain decomposition implemented using MPI. Combining the existing MPI-based decomposition with the new intranode parallelization yields a hybrid MPI/OpenMP parallel code. An extensive comparison of MPI-only ddcMD with other pure MPI codes can be found in [16].

A. Internode Operations

In typical parallel MD codes the first level of parallelism is obtained by decomposing the simulation volume into domains each of which is assigned to a compute core (*i.e.*, an MPI task). Because particles near domain boundaries interact with particles in nearby domains, internode communication is required to exchange particle data between domains. The surface-to-volume ratio of the domains and the choice of potential set the balance of communication to computation.

The domain-decomposition strategy in ddcMD allows arbitrarily shaped domains that may even overlap spatially. Also, remote particle communication between nonadjacent domains is possible when the interaction length exceeds the domain size. A domain is defined only by the position of its center and the collection of particles that it “owns.” Particles are initially assigned to the closest domain center, creating a set of domains that approximates a Voronoi tessellation. The choice of the domain centers controls the shape of this tessellation and hence the surface-to-volume ratio for each domain. The commonly used rectilinear domain decomposition employed by many parallel codes is not optimal from this perspective. The best surface-to-volume ratio in a homogeneous system is achieved if domain centers form a bcc, fcc, or hcp lattice, which are common high-density packing of atomic crystals.

In addition to setting the communication cost, the domain decomposition also controls load imbalance. Because the domain centers in ddcMD are not required to form a lattice, simulations with a non-uniform spatial distribution of particles (*e.g.*, voids or cracks) can be load balanced by an appropriate non-uniform arrangement of domain centers. The flexible domain strategy of ddcMD allows for the migration of the particles between domains by shifting the domain centers. As any change in their positions affects both load balance and the ratio of computation to communication, shifting domain centers is a convenient way to optimize the overall efficiency of the simulation. Given an appropriate metric (such as overall time spent in MPI barriers) the domains can be shifted “on-the-fly” in order to maximize efficiency [17].

B. Intranode Operations

Once particles are assigned to domains and remote particles are communicated, the force calculation begins. Figure 1(a) shows a schematic of the linked-list cell method used by ddcMD to compute pair interactions in $O(N)$ time. In this method, each simulation domain is divided into small cubic cells, and a linked-list data structure is used to organize particle data (*e.g.*, coordinates, velocities, type, and charge) in each cell. By traversing the linked list, one retrieves the information of all particles belonging to a cell, and thereby computes interparticle interactions. The dimension of the cells is determined by the cutoff length of the pair interaction, R_c .

Linked-list traversal introduces a highly irregular memory-access pattern, resulting in performance degradation. To alleviate this problem, we reorder the particles within each node at the beginning of every MD step so that the particles within the same cell are arranged contiguously in memory when the computation kernel is called. At present we choose an ordering specifically tailored to take advantage of the BlueGene “double-Hummer” (SIMD) operations. However, we could just as easily reorder the data to account for NUMA or GPGPU architectural details. We consistently find that the benefit of the regular memory access far outweighs the cost of particle ordering. The threading techniques proposed here are specifically constructed to preserve these memory-ordering advantages.

The computation within each node is described as follows. Let L_x , L_y , and L_z be the numbers of cells in the x , y , and z directions, respectively, and $\{C_k \mid 0 \leq k < L_x L_y L_z\}$ be the set of cells within each domain. The computation within each node is divided into a collection of small chunks of work called a computation unit λ . A single computation unit $\lambda_k = \{(\mathbf{r}_i, \mathbf{r}_j) \mid \mathbf{r}_i \in C_k, \mathbf{r}_j \in \text{nn}^+(C_k)\}$ for cell C_k is defined as a collection of pair-wise computations (see Fig. 1(b)), where $\text{nn}^+(C_k)$ is a set of half the nearest-neighbor cells of C_k . Newton’s third law allows us to halve the number of force evaluations and use $\text{nn}^+(C_k)$ instead of the full set of nearest-neighbor cells, $\text{nn}(C_k)$. The pairs in all computation units are unique, and thus the computation units are mutually exclusive. The set of all computation units on each node is denoted as $\Lambda = \{\lambda_k \mid 0 \leq k < L_x L_y L_z\}$. Since most of our analysis is performed at a node level, $n = N/P$ hereafter denotes the number of particles in

each node (P is the number of nodes), and p is the number of threads in each node.

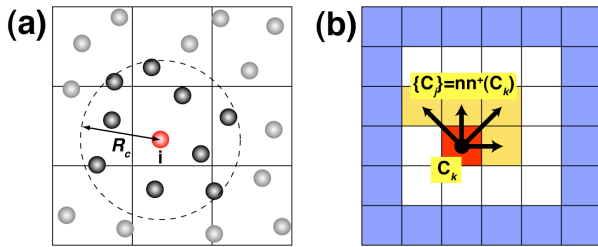


Figure 1. (a) 2D schematic of the linked-list cell method for pair computation with the cell dimension R_c . Only forces exerted by particles within the cutoff radius (represented by a two-headed arrow) are computed for particle i . (b) 2D schematic of a single computation unit λ_k . The shaded cells C_j pointed by the arrows constitute the half neighbor cells, $nn^+(C_i)$.

We parallelize the explicit pair force computation kernel of ddcMD at the thread level using OpenMP. Two major problems commonly associated with threading are: 1) race condition among threads; and 2) thread-level load imbalance [8]. The race condition occurs when multiple threads try to update the force of the same particle concurrently. Several techniques have been proposed to solve these problems:

- Duplicated pair-force computation—simple and scalable, but doubles computation. Usually used in GPGPU threading [18, 19].
- Spatial decomposition coloring [20]—scalable without increasing computation, but can cause load imbalance.
- Mutually exclusive dynamic scheduling [21, 22]—robust and suited for dynamic load balancing, but can incur considerable overhead for context switching.
- Data privatization—no penalty on computation, but with excessive $O(np)$ memory requirement per node and associated reduction sum cost [22].

This paper focuses on hybrid MPI/OpenMP parallelization on Sequoia, the third generation of BlueGene, which will be online in 2011-2012 at Lawrence Livermore National Laboratory (LLNL). On this SMP platform called Sequoia, MPI-only programming will not be an option for full-scale runs with 3.2 million concurrent threads.

III. DATA-PRIVATIZATION SCHEDULING ALGORITHM

A traditional data-privatization algorithm avoids write conflicts by replicating the entire write-shared data structure and allocating a private copy to each thread (Fig. 2(a)). The memory requirement for this redundant allocation scales as $O(np)$. Each thread computes forces for each of its computation units and stores the force values in its private array instead of the global array. This allows each thread to compute forces independently without a critical section. After the force computation for each MD step is completed, the private force arrays are reduced to obtain the global forces.

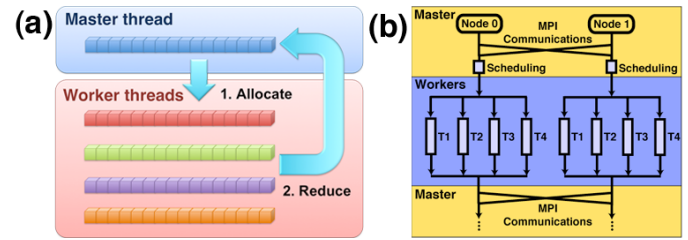


Figure 2. (a) Schematic of a memory layout for a traditional data privatization. (b) Schematic workflow of our hybrid MPI/OpenMP scheme.

To reduce the redundant memory requirement, we have developed a low-overhead approach that provides excellent load balancing while imposing minimal interference on the worker threads. Our algorithm utilizes a scheduler to distribute the workload before entering the pair computation, *i.e.*, parallel section. In this approach, the scheduling cannot interfere with the worker threads since the scheduling is already completed before the worker threads are started. Because the schedule is recomputed every MD step (or perhaps every few MD steps) there is adequate flexibility to adapt load balancing to the changing dynamics of the simulation.

The hybrid MPI/OpenMP parallelization of ddcMD is implemented by introducing the thread scheduler into the MPI-only ddcMD. Figure 2(b) shows the workflow of the hybrid MPI/OpenMP code. The program repeats the following computational phases: First, the master thread performs initialization and internode communications using MPI; the scheduler computes the scheduled workload for each thread; and the worker threads execute the workloads in an OpenMP parallel section.

Since the scheduling is performed frequently, the load-balancing algorithm needs to be simple yet provide sufficient load-balancing capability. Therefore, we have adopted a greedy approach for the load-balancing scheduler, which is discussed and analyzed in section III-A. In section III-B, the load-balancing scheduler is further enhanced by introducing a nucleation-growth allocation algorithm.

A. Thread-Level Load-Balancing Algorithm

We implement thread-level load balancing based on a simple greedy approach, *i.e.*, iteratively assign a computation unit to the least-loaded thread, until all computation units are assigned. Let $T_i \subseteq \Lambda$ denote a mutually exclusive subset of computation units assigned to the i -th thread, $\cap_{k=0, \dots, LxLyLz} \lambda_k = \emptyset$. The computation time spent on λ_k is denoted as $\tau(\lambda_k)$. Thus, the computation time of each thread $\tau(T_i)$ is a sum of all computation units assigned to thread T_i . The algorithm initializes T_i to be empty, and loops over λ_k in Λ . Each iteration selects the least-loaded thread $T_{\min} = \text{argmin}(\tau(T_i))$, and assigns λ_k to it. This approach is a 2-approximation algorithm [23] and its pseudo code is shown in Fig. 3.

Algorithm Fine-Grain Load Balancing

-
1. **for** $0 \leq i < p$ **do**
 2. $T_i \leftarrow \emptyset$
 3. **end do**
 4. **for each** λ_k **in** Λ **do**
 5. $T_{\min} \leftarrow \operatorname{argmin}_{0 \leq i < p} (\tau(T_i))$
 6. $T_{\min} \leftarrow T_{\min} \cup \lambda_k$
 7. **end do**
-

Figure 3. Thread load-balancing algorithm.

The algorithm is simple yet provides an excellent load-balancing capability. As shown below, this approach has a well-defined upper bound on the load imbalance. To quantify the load imbalance, we define a load-imbalance factor γ as the difference between the runtime of the slowest thread and the average runtime,

$$\gamma = \frac{\max(\tau(T_i)) - \tau_{\text{average}}}{\tau_{\text{average}}}. \quad (1)$$

By definition, $\gamma = 0$ when the loads are perfectly balanced. Since $\min(\tau(T_i)) \leq \tau_{\text{average}}$,

$$\gamma \leq \frac{\max(\tau(T_i)) - \min(\tau(T_i))}{\tau_{\text{average}}}. \quad (2)$$

In our load-balancing algorithm, the workload of T_{\min} is increased at most by $\max(\tau(\lambda_k))$ at each iteration. This procedure guarantees that the variance of the workloads among all threads is limited by

$$\max(\tau(T_i)) - \min(\tau(T_i)) \leq \max(\tau(\lambda_k)). \quad (3)$$

Substituting Eq. (3) in Eq. (2) provides an upper limit for the load-imbalance factor,

$$\gamma \leq \frac{\max(\tau(\lambda_k))}{\tau_{\text{average}}}. \quad (4)$$

Performance of this load-balance scheduling algorithm depends critically on the knowledge of time spent on each computation unit $\tau(\lambda_k)$. Since the runtime of the computation units are unknown to the scheduler prior to the actual computation, the scheduler has to accurately estimate the workload of each computation unit. Fortunately, since particle positions change slowly $\tau(\lambda_k)$ remains highly correlated between the consecutive MD steps. Therefore, we use $\tau(\lambda_k)$ measured in the previous MD step as an estimator of $\tau(\lambda_k)$. This automatically takes into account any local variations in the cost of the potential evaluation. For the first step as well as steps when the cell structure changes significantly (e.g., redistribution of the domain centers), the workload of cell $\tau(\lambda_k)$ is estimated by counting the number of pairs in λ_k .

B. Load Balancing Spanning Forest Partitioning Algorithm

As mentioned before, the memory requirement of the data-privatization algorithm is $O(np)$. However, since only a small subset of Λ is assigned to each thread it is not necessary to allocate a complete copy of the force array for each thread. Therefore, we allocate only the necessary portion of the global force array corresponding to the computation units assigned to each thread as a private force array. This idea is embodied in a three-step algorithm (Fig. 4): 1) the scheduler assigns computation units to threads and then determines which subset of the global data each thread requires; 2) each thread allocates its private memory as determined by the scheduler; 3) private force arrays from all threads are reduced into the global force array.

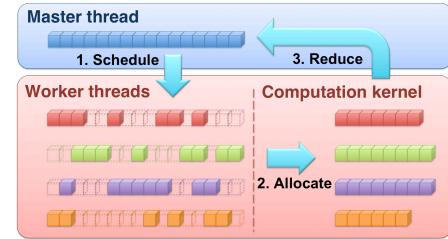


Figure 4. Memory layout and three-step algorithm for scheduling algorithm.

To do this, we create a mapping table between the global force-array index of each particle and its thread-array index in a thread memory space. Since ddcMD sorts the particle data based on the cell they reside in, only the mapping from the first global particle index of each cell to the first local particle index is required. The local ordering within each cell is identical in both the global and private arrays.

It should be noted that assigning computation unit λ_k to thread T_i requires memory allocation more than the memory for the particles in C_k . Since each computation unit computes the pair forces of particles in cell C_k and half of its neighbor cells $\text{nn}^+(C_k)$ as shown in Fig. 1(b), the force data of particles in $\text{nn}^+(C_k)$ need to be allocated as well. In order to minimize the memory requirement of each thread, the computation units assigned to it must be spatially proximate, so that the union of their neighbor-cell sets has a minimal size. This is achieved by minimizing the surface-to-volume ratio of T_i . For this purpose, we implement the LBSF algorithm (Fig. 5) by modifying algorithm shown in section III-A. First, we randomly assign a root computation unit to each thread. Then, the iteration begins by selecting the least-loaded thread T_{\min} . From the surrounding volume of T_{\min} , we select a computation unit λ_{j^*} that has the minimum distance to the centroid of T_{\min} , and add it to T_{\min} . The algorithm repeats until all computation units are assigned. If all of the surrounding computation units of T_{\min} are already assigned, T_{\min} randomly chooses a new unassigned computation unit as a new cluster's root and continue to grow from that point.

Figure 6 shows a 2D example of the LBSF partitioning. Figure 6(a) shows non-uniform particle distribution, where the workload in each cell is assumed to be proportional to the number of particles in the cell. Figure 6(b) illustrates the result of a partitioning by scheduler. Most computation units on the

lower left corner are assigned to T_1 , while the rest are assigned to T_2 . The load-imbalance factor γ in this example is 0.044.

Algorithm LBSF Partitioning

```

1.  $i \leftarrow 0$ 
2. while  $i < p$  do
3.   repeat
4.      $\lambda_{\text{root}} \leftarrow \text{random}(\Lambda)$ 
5.     until  $\lambda_{\text{root}} \notin T_{j^{(i)}}$ 
6.      $T_i \leftarrow \lambda_{\text{root}}$ 
7.      $i \leftarrow i + 1$ 
8.   end do
9.   while  $\cup_{0 \leq i < p} T_i \neq \Lambda$  do
10.     $T_{\min} \leftarrow \text{argmin}_{0 \leq i < p} (\tau(T_i))$ 
11.     $j^* \leftarrow \text{argmin}_{C_j \in \text{nn}(T_{\min})} (\|\text{centroid}(T_{\min}) - C_j\|)$ 
12.     $T_{\min} \leftarrow T_{\min} \cup \lambda_{j^*}$ 
13.   end do

```

Figure 5. LBSF partitioning algorithm.

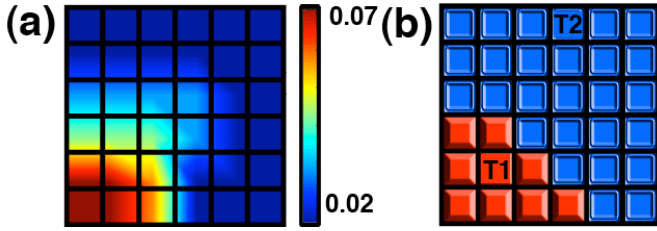


Figure 6. 2D illustration of the LBSF partitioning algorithm. (a) Spatial particle distribution where the normalized particle density is color-coded. (b) The corresponding computation-unit assignment to two threads originating at T_1 and T_2 cells.

The memory requirement of this algorithm can be analyzed as follows. The memory for each thread comes from two sources: 1) memory for the actually assigned computation units M_λ ; and 2) memory for the surface cells neighboring the assigned computation units M_s . The amount of memory requirement for one thread is

$$\begin{aligned} M_\lambda &= O(n/p) \\ M_s &= O(n/p)^{2/3}, \end{aligned} \quad (5)$$

and the memory footprint of p threads on a single node is

$$\begin{aligned} M_{\text{node}} &= p(M_\lambda + M_s) \\ &= O(p(n/p) + p(n/p)^{2/3}). \\ &= O(n + p^{1/3}n^{2/3}) \end{aligned} \quad (6)$$

The asymptotic memory requirement for each node is thus $O(n + p^{1/3}n^{2/3})$, which is much smaller than the $O(np)$ memory requirement of traditional data-privatization.

Although this algorithm reduces the memory footprint significantly, it poses a minor difficulty in the reduction sum. This difficulty arises from the fact that the partial private force arrays are not aligned with each other. Nevertheless, the cost of linear reduction sum is reduced to $O(n + p^{1/3}n^{2/3})$ as a consequence of the reduced memory footprint. In fact, for a

given p , the computation time of the partial linear reduction sum could be less than that of a hypercube reduction when n is large such that $O(p^{1/3}n^{2/3}) < O(n \log p)$.

IV. PERFORMANCE EVALUATION

In this section, we perform performance measurements and analysis of the algorithm described in the previous section. Section IV-A measures the load-imbalance factor of our scheduler, and section IV-B measures the memory-footprint reduction by our approach, confirming its $O(n + p^{1/3}n^{2/3})$ scaling. Section IV-C demonstrates the reduction of the scheduling cost without affecting the quality of load balancing, followed by strong-scaling comparison of the hybrid MPI/OpenMP and MPI-only schemes.

A. Thread-Level Load Balancing

We perform a load-balancing test for the scheduling algorithm on a dual six-core AMD Opteron 2.3 GHz with $n = 8,192$ (Fig. 7(a)). The actual measurement of the load-imbalance factor γ is plotted as a function of p , along with its estimator introduced in section III-A and the theoretical bound, Eq. (4). The results show that $\gamma_{\text{estimated}}$ and γ_{actual} are close, and are below the theoretical bound. γ is an increasing function of p , which indicates the severity of the load imbalance for a highly multi-threaded environment and highlights the importance of the fine-grain load balancing.

We also observe that the performance fluctuates slightly depending on a selection of root nodes in LBSF algorithm. While the random root selection tends to provide robust performance compared to deterministic selection, it is possible to use some optimization techniques (e.g., reinforcement learning) to dynamically optimize the initial cell selection at runtime. For more irregular applications, it is conceivable to combine the light-overhead thread-level load balancing in this paper with a high quality node-level load balancer such as a hypergraph-based approach [24].

B. Memory Footprint

To test the memory efficiency of the proposed method, we perform simulations on a four quad-core AMD Opteron 2.3 GHz machine with a fixed number of particles $n = 8,192$, 16,000, and 31,250. We measure the memory allocation size for 100 MD steps while varying the number of threads p from 1 to 16. Figure 7(b) shows the average memory allocation size of the force array as a function of the number of threads for the proposed algorithm compared to that of a traditional data-privatization algorithm. The results show that the memory requirement for 16 threads is reduced by 65%, 72%, and 75%, respectively, for $n = 8,192$, 16,000, and 31,250 compared with the traditional $O(np)$ memory per-node requirement. In Fig. 7(b), the dashed curves show the reduction of memory requirement per thread estimated as

$$m = ap^{-1} + bp^{-2/3}, \quad (7)$$

where the first term represents the memory scaling from actual assigned cells and the second term represents scaling from

surface cells of each thread, see Eq. (5). The regression curves fit the measurements well, indicating that the memory requirement is accurately modeled by $O(n+p^{1/3}n^{2/3})$.

We also measure the computation time spent for the reduction sum of the private force arrays to obtain the global force array. Figure 7(c) shows the reduction-sum time as a function of the number of threads p for $n = 8,192, 16,000,$ and $31,250$ particles. Here, dashed curves represent the regression,

$$t_{\text{reduction}} = ap^{1/3} + b, \quad (8)$$

which fit all cases well.

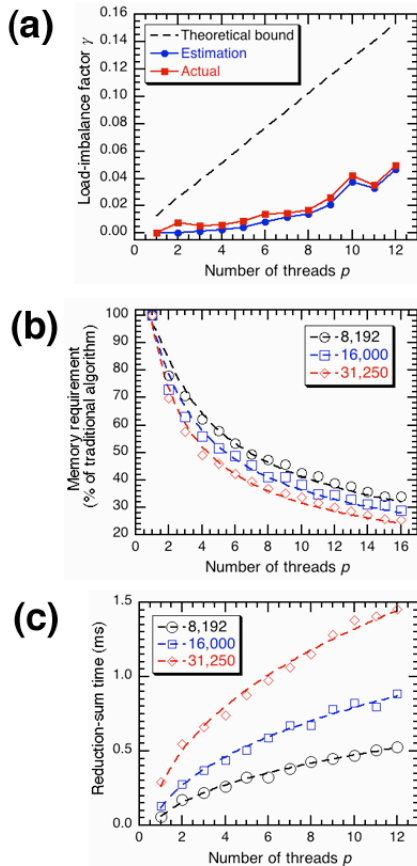


Figure 7. (a) Load-imbalance factor γ as a function of p from theoretical bound, scheduler estimation, and actual measurement. (b) Average memory consumption for the private force arrays as a function of p using LBSF compared to the conventional method. Numbers in the legend denote n . (c) Average reduction-sum time of the LBSF scheduling as a function of p .

C. Strong-Scaling Performance

We measure the performance of the combined nucleation-growth allocation and load-balancing algorithms. Figure 8(a) shows the thread-level strong-scaling speedup on a four quad-core AMD Opteron 2.3 GHz. The algorithm achieves a speedup of 14.43 on 16 threads, *i.e.*, the strong-scaling multi-threading parallel efficiency is 0.90. As shown in section IV-B, the combined algorithm reduces the memory consumption up to 65% for $n = 8,192$, while still maintaining excellent strong scalability.

Next, we compare the strong-scaling performance of the hybrid MPI/OpenMP and MPI-only schemes for large-scale problems on BlueGene/P at LLNL. One BlueGene/P node consists of four PowerPC 450 850 MHz processors. The MPI-only implementation treats each core as a separate task, while the hybrid MPI/OpenMP implementation has one MPI task per node, which spawns four worker threads for the force computation. The test is performed on $P = 8,192$ nodes, which is equivalent to 32,768 MPI tasks in the MPI-only case and 32,768 threads for hybrid MPI/OpenMP. Figures 8 (b) and (c) show the running time of 843,750 and 1,687,500 particles systems for the total number of cores ranging from 1,024 to 32,768. The result indicates that the hybrid scheme performs better when the core count is larger than 8,192. On the other hand, the MPI-only scheme gradually stops gaining benefit from the increased number of cores and becomes slower. The MPI/OpenMP code shows 2.58 \times and 2.16 \times speedup over the MPI-only implementation for $N = 0.84$ and 1.68 million, respectively, when using 32,768 cores. Note that the crossover granularity of the two schemes is $n/p \sim 100$ particles/core for both cases.

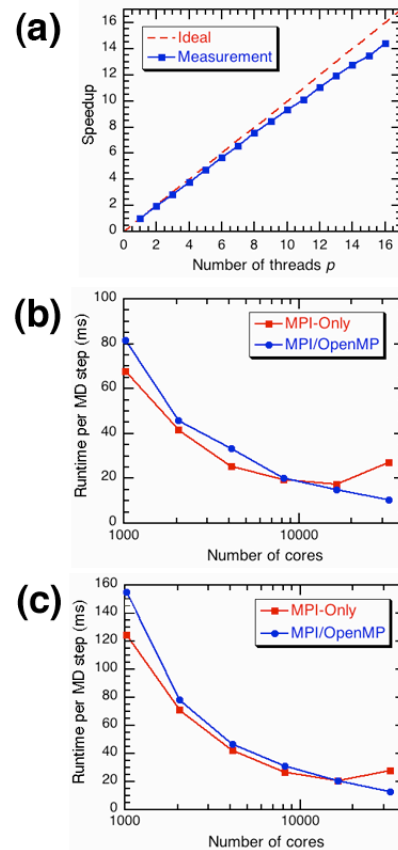


Figure 8. (a) Thread-level strong scalability of the parallel section on a four quad-core AMD Opteron 2.3 GHz with fixed problem size at $n = 8,192$ particles. (b) Total running time per MD steps of 1,024 – 32,768 Power PC 450 850 MHz cores on BG/P for a fixed problem size at $N = 0.84$ -million particles and (c) 1.68-million particles.

This result indicates that the Amdahl's law limits the performance of the MPI/OpenMP code when using small number of nodes. Namely, only the pair kernel is parallelized, while the rest of the program is sequential in the thread level. This disadvantage of the MPI/OpenMP code diminishes as the number of cores increases. Eventually, the hybrid MPI/OpenMP code performs better than the MPI-only code after 8,192 cores. The main factors underlying this result are: 1) the surface-to-volume ratio of the MPI-only code is larger than that of the hybrid MPI/OpenMP code; and 2) the communication latency for each node of the MPI-only code is four times larger than that of the hybrid MPI/OpenMP code. This result confirms the assertion that the MPI/OpenMP model (or similar hybrid schemes) will be required to achieve better strong-scaling performance on large-scale multicore architectures.

V. CONCLUSIONS

Our LBSF partitioning algorithm successfully overcomes the disadvantages of the traditional data-privatization threading with minimal overhead. The LBSF scheduling guarantees a bounded load imbalance while reducing the memory requirement from $O(np)$ to $O(n+p^{1/3}n^{2/3})$. The cost of scheduling can be eliminated without the loss of load-balancing quality by reducing the scheduling frequency. Also, benchmarks of the massively parallel MD simulations suggest significant performance benefits of the hybrid MPI/OpenMP scheme for fine-grain large-scale applications.

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-469312). The work at USC was partially supported by DOE BES/EFRC/SciDAC/SciDAC-e/INCITE and NSF PetaApps/EMT.

REFERENCES

- [1] J. C. Phillips, *et al.*, "NAMD: Biomolecular simulations on thousands of processors," in *Supercomputing*, Los Alamitos, CA, 2002.
- [2] F. H. Streitz, *et al.*, "Simulating solidification in metals at high pressure: The drive to petascale computing," *SciDAC 2006: Scientific Discovery Through Advanced Computing*, vol. 46, pp. 254-267, 2006.
- [3] K. J. Bowers, *et al.*, "Zonal methods for the parallel execution of range-limited N-body simulations," *Journal of Computational Physics*, vol. 221, pp. 303-329, 2007.
- [4] B. Hess, *et al.*, "GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," *Journal of Chemical Theory and Computation*, vol. 4, pp. 435-447, 2008.
- [5] D. E. Shaw, *et al.*, "Millisecond-scale molecular dynamics simulations on Anton," in *Supercomputing*, Portland, Oregon, 2009, pp. 1-11.
- [6] K. Nomura, *et al.*, "A metascalable computing framework for large spatiotemporal-scale atomistic simulations," in *International Parallel and Distributed Processing Symposium*, 2009.
- [7] J. N. Glosli, *et al.*, "Extending stability beyond CPU millennium: a micron-scale atomistic simulation of Kelvin-Helmholtz instability," in *Supercomputing*, Reno, Nevada, 2007, pp. 1-11.
- [8] S. R. Alam, *et al.*, "Impact of multicores on large-scale molecular dynamics simulations," in *International Parallel and Distributed Processing Symposium*, Miami, Florida USA, 2008.
- [9] L. Peng, *et al.*, "A scalable hierarchical parallelization framework for molecular dynamics simulation on multicore clusters," in *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, 2009.
- [10] M. J. Chorley, *et al.*, "Hybrid message-passing and shared-memory programming in a molecular dynamics application on multicore clusters," *International Journal of High Performance Computing Applications*, vol. 23, pp. 196-211, 2009.
- [11] R. Rabenseifner, *et al.*, "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes," *Proceedings of the Parallel, Distributed and Network-Based Processing*, pp. 427-436, 2009.
- [12] C. Long, *et al.*, "Dynamic load balancing on single- and multi-GPU systems," in *International Parallel and Distributed Processing Symposium*, 2010, pp. 1-12.
- [13] D. York and W. Yang, "The fast Fourier Poisson method for calculating Ewald sums," *The Journal of Chemical Physics*, vol. 101, pp. 3298-3300, 1994.
- [14] R. Hockney and J. Eastwood, *Computer simulation using particles*. New York: McGraw-Hill, 1981.
- [15] T. Darden, *et al.*, "Particle mesh Ewald: An N log(N) method for Ewald sums in large systems," *The Journal of Chemical Physics*, vol. 98, pp. 10089-10092, 1993.
- [16] D. F. Richards, *et al.*, "Beyond homogeneous decomposition: scaling long-range forces on Massively Parallel Systems," in *Supercomputing*, Portland, Oregon, 2009, pp. 1-12.
- [17] J.-L. Fattbert, *et al.*, unpublished.
- [18] A. Sunarso, *et al.*, "GPU-accelerated molecular dynamics simulation for study of liquid crystalline flows," *Journal of Computational Physics*, vol. 229, pp. 5486-5497, 2010.
- [19] J. Yang, *et al.*, "GPU accelerated molecular dynamics simulation of thermal conductivities," *Journal of Computational Physics*, vol. 221, pp. 799-804, 2007.
- [20] C. Hu, *et al.*, "Efficient parallel implementation of molecular dynamics with embedded atom method on multi-core platforms," in *International Conference on Parallel Processing Workshops*, 2009, pp. 121-129.
- [21] D. W. Holmes, *et al.*, "An events based algorithm for distributing concurrent tasks on multi-core architectures," *Computer Physics Communications*, vol. 181, pp. 341-354, 2010.
- [22] K. Madduri, *et al.*, "Memory-efficient optimization of Gyrokinetic particle-to-grid interpolation for multicore processors," in *Supercomputing*, Portland, Oregon, 2009, pp. 1-12.
- [23] J. Kleinberg and E. Tardos, *Algorithm Design*, 2 ed.: Pearson Education, Inc., 2005.
- [24] U. V. Catalyurek, *et al.*, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *International Parallel and Distributed Processing Symposium*, 2007, pp. 1-11.

Efficient data access for Open Modeling Interface (OpenMI) components

Tom Bulatewicz, Daniel Andresen

Dept. of Computing and Information Sciences, Kansas State University, Manhattan, KS, USA

Abstract—Data management for linked (or coupled) simulation models can be a challenging task when deploying to grid environments. In cases where the linked models conform to a standard interface for data input and output, general-purpose data providers can be used to supply data to the models from online sources, reducing the complexity of the deployment. We have developed a data provider component that conforms to the Open Modeling Interface (OpenMI) that is suitable for use on computational grids. Through the application of three techniques, caching, prefetching, and pipelining, the component efficiently retrieves data from standards-based web services and delivers the data to OpenMI-compliant models. Each technique resulted in varying performance improvements both within a single simulation and across multiple simulations concurrently executing on a cluster. In this paper we report on the design of the component and the evaluation of its performance.

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

1. Introduction

Computer models require input data in order to perform simulations. This data may originate from a variety of sources, may vary in space and time, and may be used during the initialization of a simulation and during its execution. In the case of linked (or coupled) models that execute independently and cooperate to collectively perform a simulation, each model requires its own input data. Models often have unique input data formats requiring inputs to be individually prepared for each model prior to the simulation run and often results in duplication of the data that is common between models. These input datasets must be deployed with the models to the execution environment, such as a computational grid. To obviate the need to prepare input data in model-specific formats and to increase the portability of datasets between models, standard data formats have been developed (e.g. netCDF [1]). To obviate the need to deploy datasets to the execution environment and to provide access to real time measurement data, data distribution frameworks have been developed that allow models to access online data sources (e.g. via web services). In the general case, models must have the capability to use these standard data formats and data distribution frameworks. In the case of models that are software components with well-defined

input/output interfaces (e.g. CCA [2]), data access using these standards can be implemented in general-purpose data components which can be linked to model components. Such data provider components play an important role in any linked modeling environment.

The Open Modeling Interface (OpenMI) [3] provides 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 corresponds to the exchange items) at runtime.

The *GetValues* function has three parameters and returns a set of values as illustrated in Figure 1. The parameters

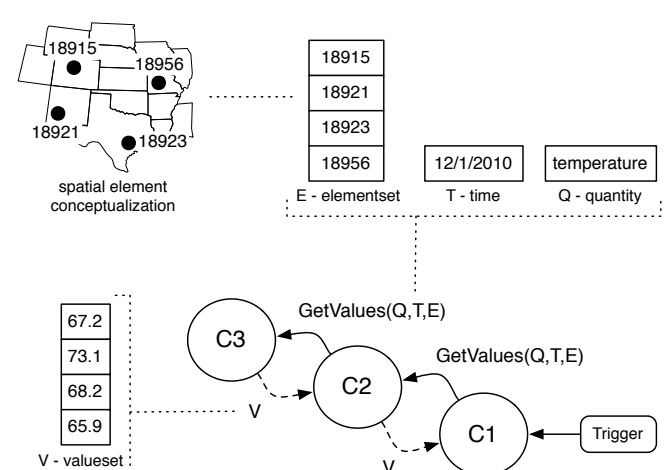


Fig. 1: OpenMI pull-based execution. Solid lines indicate function calls and dashed lines indicate the flow of data.

collectively identify a specific set of values that represent the state of a quantity at a single point in time over some spatial distribution. The quantity is described by a textual identifier, the time by a modified-Julian date, and the spatial distribution by a list of spatial elements called an *elementset*. An elementset is a list of geo-referenced spatial elements such as a point, line segment, or polygon. The *GetValues* function returns a list of floating-point values called a *valueset*. The values in a valueset describe the state of a single quantity at a specific point in time where each individual value corresponds to a different spatial element (based on array index).

The *GetValues* function not only provides a means for the exchange of data between a set of linked components (called a *composition*) but it also provides a means for their coordinated execution at runtime. A special component called a *trigger* begins by invoking *GetValues* on one of the components. The first time *GetValues* is invoked on a component it begins executing (e.g. performing time steps) from its starting simulation time. The component executes until it requires input values for one of the quantities on one of its links, at which point it invokes *GetValues* on the component at the providing end of the link and blocks. The component blocks until the call to *GetValues* completes and the values are returned, at which point it continues its execution. The components take turns executing and *pull* data from each other until the simulation completes. A component only performs time steps as-needed in direct response to a call to its *GetValues* function.

Compositions can be created in a highly automated way. Using visual software tools, a scientist chooses a set of components of interest, interactively specifies the links between them, and then executes the simulation. Each *link* maps an output exchange quantity from one component to the input exchange quantity of another component and links can be uni-directional or bi-directional.

In this work we present the design and evaluation of a general-purpose Data Provider Component (DPC) that is capable of delivering data from online sources to OpenMI components. We describe the design and implementation of the DPC 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

2.1 System Overview

Figure 2 illustrates the movement of data through a distributed data delivery system for linked model components. Compositions of linked components execute on cluster nodes. Each composition includes a DPC that retrieves data from web services and provides it to the other components. DPCs within compositions that are running on different cluster nodes share data with each other. When a component

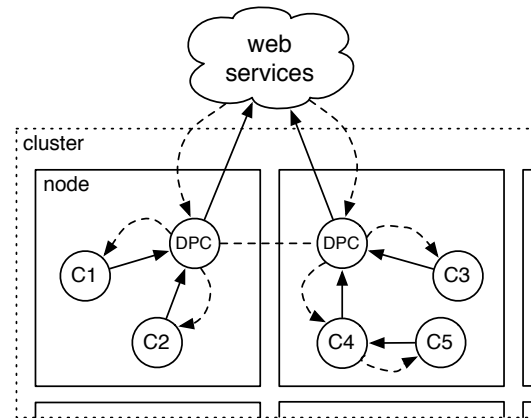


Fig. 2: System overview.

needs input values it invokes *GetValues* on the DPC for the needed quantity, time, and spatial elements and the DPC returns the appropriate valueset. The DPC calls web services for a specific quantity identifier, time, and list of location identifiers (which correspond to elements) and then extracts the valueset from the response. Thus the content of a web service call mirrors that of a *GetValues* call. We assume that each spatial element has a unique identifier, although this is not enforced by the OpenMI. Any web service that can be queried for a quantity, time, and list of locations and returns a list of values could be used as a data source for the DPC.

Within the context of water resources, there are several standards for data models to store observations data and web services to access them [4]. The WaterOneFlow [5] web service API has recently been utilized by several government agencies (<http://hiscentral.cuahsi.org>) and has a *GetValues* method that can be queried for an individual quantity in a single location or region for a timespan. Time series data is returned in XML that conforms to the WaterML schema [6].

The initial implementation of the DPC supports SOAP calls to WaterOneFlow web services and to a custom variant that allows multiple quantities and locations to be queried in a single call which is not currently supported by the WaterOneFlow API but is necessary to evaluate the performance of the DPC. The implementation supports parsing WaterML responses via SAX. An example of a request and response is shown in Figure 3. The DPC's configuration file defines its output exchange items (i.e. quantities and elementsets) and the information about each web service, such as the URL, API, response format, and which quantities can be queried. The implementation can be extended to support other web services and response formats.

If the DPC were to make a web service call on each invocation of *GetValues* then the rest of the composition would be paused for the duration of the call, adding to the overall runtime of the simulation. In addition, invocations by different components for the same values would result in

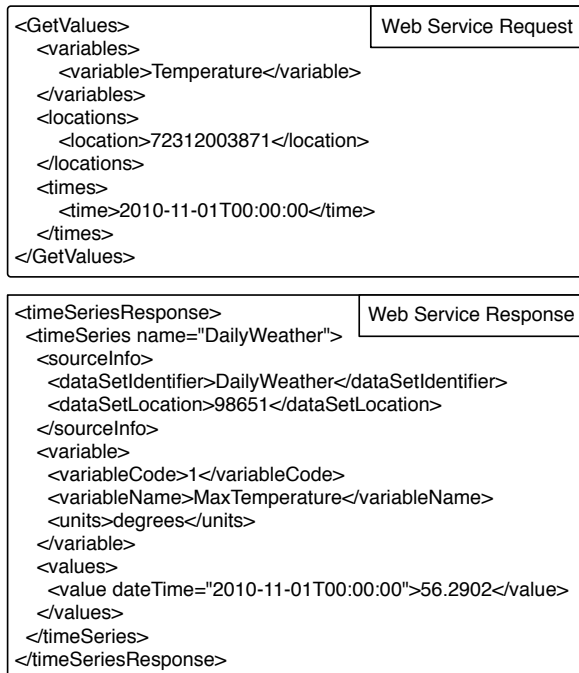


Fig. 3: Example web service request and response (attributes and SOAP envelope removed for clarity).

duplicate web service calls which would also increase the overall runtime and inefficiently utilize network bandwidth and other resources. To minimize the effect that the DPC has on the execution of the composition in terms of runtime and resource use, the DPC must (1) minimize the time it takes for `GetValues` to return a valueset and (2) minimize the number of times a valueset is retrieved from a web service.

In the ideal case there would be zero wait time and each valueset would be retrieved from a web service once. To these ends, the DPC utilizes three strategies: caching, prefetching, and pipelining. All valuesets are retrieved from web services once and are then cached so that they are immediately available for subsequent invocations of `GetValues` by other components (within and across compositions) and subsequent executions of the composition. Since components typically advance forward through simulation time, valuesets are prefetched so that they are available in the cache before they are requested. Multiple web service calls are performed simultaneously in a pipelined fashion to maximize use of available network bandwidth.

The DPC consists of a fetching module and a caching module as illustrated in Figure 4. The fetching module identifies the valuesets that the other components need, retrieves them from the web services, and then stores them in the cache. The caching module handles calls to `GetValues` by retrieving the appropriate values from the cache, assembling the valueset, and returning it to the calling component.

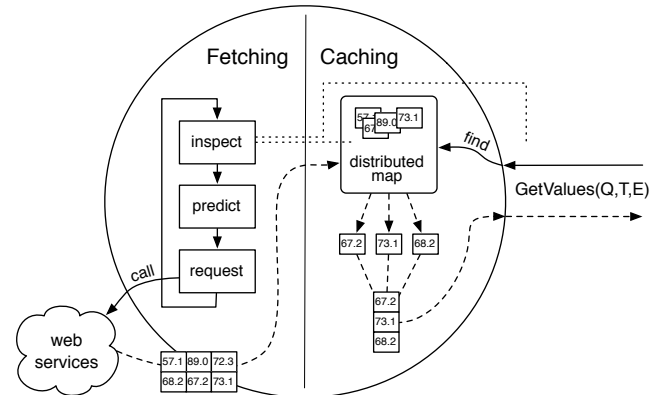


Fig. 4: Operation of the data provider component. Solid lines indicate function calls and dashed lines indicate the flow of data.

2.2 Caching

During the execution of a composition, several components within a single composition may request the same valuesets from a DPC. Components in independently executing compositions on different cluster nodes may request the same valuesets from different DPCs. In both cases it is advantageous for the DPCs to cache the valuesets that they retrieve from the web services and to share those valuesets across all the DPCs that are executing simultaneously in different compositions across a cluster. The same valuesets may be needed on subsequent executions of the same composition so it is also advantageous for the cached valuesets to be persisted between executions.

To serve these needs, a clustering, scalable data distribution platform (Hazelcast [7]) is utilized by the caching module to store the values retrieved from the web services. Each DPC has an instance of the platform peer that is managed by a set of threads within the same process as the DPC. Instances dynamically cluster and discover peers via multicast and communicate via TCP/IP. Thus, there are no servers involved and each DPC is self-sufficient and shares data directly with other DPCs. The data structure used to store the values is a distributed map. Entries in the cache are evenly partitioned onto the currently executing instances across the cluster (i.e. DPCs). Each instance uses a private database file [8] to persist the cache entries between executions.

The valuesets that are retrieved from the web services are decomposed into individual values and stored in the distributed map. Storing the individual values allows the map to assemble different valuesets ad-hoc as they are requested by model components, maximizing the reusability of data and the effectiveness of the cache. Alternatively the complete valuesets could be stored as single entries in the map but this would limit the reusability of the data to cases where subsequent requests are for the same combination of

quantity, time, and elementset.

The DPC may be linked to several model components that use different time steps, different input element sets, and different units. It is the responsibility of the providing component (the DPC) to apply necessary transformations (units, spatial, temporal) to meet the input of the requesting component (the model). To maximize the effectiveness of the cache the DPC caches the values retrieved from the web service and when a component requests a valueset the appropriate values are extracted from the cache, transformed, and provided to the model. Alternatively the DPC could cache the transformed valuesets but this would limit the reusability of the data to cases where subsequent requests are for the same combination of quantity, time, elementset, and units.

Entries in the distributed map are tuples of the form $\langle \text{value}, \text{availability} \rangle$ and are keyed by a textual string that uniquely identifies a value by the concatenation of its time, quantity identifier, and element identifier. The value is a floating point number and the availability flag is a boolean that indicates whether the value has been retrieved (true) or a web service call for the value is in progress (false). The contents of the map are persisted to the database upon completion of a composition run and then restored into the map upon startup.

When `GetValues` is called by a component, the DPC checks to see if the value for each element of the requested elementset exists in the cache by creating the key and then performing a get operation on the distributed map using the key. If values for any of the elements are missing then the valueset is not available and the cache waits for a period of time before checking again (during which the composition is blocked). The cache relies on the fetching module to populate the map with values.

2.3 Fetching

The simulation of physical processes (especially those for which the OpenMI was initially designed) typically involve the calculation of output quantities over a simulation period. The components step through simulation time and periodically request values from each other. The frequency at which `GetValues` is invoked on the DPC by other components is likely not the most efficient frequency for the DPC to call the web services. For this reason the DPC prefetches valuesets to minimize the time that the other components must wait when calling `GetValues` on the DPC. Multiple web service calls are issued simultaneously in a pipelined fashion to take advantage of multi-core and multi-host web services.

Throughout the execution of a composition the components are at approximately the same point in simulation time. This is because components typically require input data that reflects their current simulation time which requires that the components providing the inputs advance to that same point in simulation time. Prefetching is thus most effective when

it is done such that the data for all components is prefetched to the same future point in simulation time.

Prefetching relies on knowledge of what data will be needed before it is requested. It is not possible for the DPC to obtain this information directly from the other components as this functionality is not supported by the OpenMI. The DPC predicts what valuesets will be requested in the future by observing what valuesets have been requested in the past. Components that use a fixed-length time step request data from the DPC at fixed intervals making it possible for the DPC to identify these components and determine the length of their time steps. In such cases the DPC can accurately predict the valuesets that will be requested in the future. It is more difficult for the DPC to predict the data needs of components that use a variable-length time step and is not addressed in this work.

Web service calls request different combinations of quantity, time, and spatial elements, and thus these calls may be coalesced along any or all of these three dimensions. Since our goal is to minimize wait time, the coalescing strategy should group together similar requests as much as possible without inducing additional wait time. Of the three dimensions, only spatial coalescing is guaranteed to not incur any unneeded wait time because the DPC cannot provide a valueset to a model component unless all values for the entire elementset have been retrieved. Coalescing by time or quantity may result in a model component waiting a longer period of time for a valueset that is part of a larger request than it would have if the valueset was requested individually. For this reason the DPC only coalesces web service requests spatially such that each valueset, corresponding to a single quantity and time over a complete set of spatial elements, is requested in each web service call.

2.3.1 Runtime Operation

The fetching module manages a *fetch thread* for each web service. It is responsible for identifying the valuesets that must be retrieved from the web service and issuing the web service calls to retrieve them. When the state of the fetch thread changes (as a result of a call starting or completing) or a cache miss occurs, the fetch thread attempts to identify and download as many valuesets as possible. A single quantity may be provided along several links at different temporal intervals or different spatial elementsets, so each link must be checked for necessary valuesets individually. Each attempt makes a series of passes through the links until no valuesets are found or until there are no available resources. On each pass, the earliest valueset (either by request or predicted) needed by a link that is not already in the cache is identified and a web service call is started (at which time placeholder entries (availability = false) for the values are created in the cache to indicate that they are being retrieved). Multiple DPCs may request intersecting valuesets in which case only partial valuesets are requested from the web service.

Resource usage maximums can be externally parameterized statically in terms of the maximum number of concurrent web service calls and dynamically in terms of maximum network bandwidth or CPU utilization (via the Java Management Extensions). When the maximum number of concurrent calls is met or the maximum CPU or network bandwidth is reached, prefetching stops for a period of time before it is attempted again.

To keep all links prefetched to approximately the same point in simulation time, each link is prefetched up to a moving limit: $limit = \min\{p + n \times i, e\}$ where p is the earliest time to which all links are prefetched to, i is the longest request interval across all links, and e is the ending time of the composition. The constant n controls how close in simulation time the links are to be prefetched to and should be 1 when the difference in request intervals is large and may be higher when the difference is small.

3. Experimental Results

We conducted a performance study using an onsite Linux-based Beowulf cluster. The compute nodes had 16-core 2 GHz processors and 64 GB of memory and the server node had a 4-core 2.7 GHz processor and 8 GB of memory connected via gigabit ethernet. The software components were implemented in Java (Alterra SDK) conforming to version 1.4 of the OpenMI and the web service was implemented in ASP.NET. The time spent by the web service to generate the response to each request was configurable and the contents of the response contained random numbers.

To represent a model component we created a placeholder component that used a fixed-length time step of 1 day and would sleep for 10 s between time steps to mimic the time spent calculating a time step, which we call the *processing time*. There was a single link between the model component and the DPC and the elementset consisted of 50000 elements. Each composition ran for 60 time steps resulting in a total of 3 million values in the cache at the end of the run.

3.1 Caching

To investigate the effect of the cache we performed two sets of simulations with varying numbers of model components with and without the cache and measured the runtime and amount of data transferred (prefetching was disabled). In the first set, the model components were part of a single composition and linked to a single DPC. In the second set, each model component was in a separate composition running on a different node so each composition consisted of one model component and one DPC. In both sets the model components request the exact same valuesets to maximize the effect of the cache. Since the design of the DPC necessitates a cache, the effect of disabling the cache was approximated by removing each valueset from the cache after it was returned to a model component.

When a component invokes `GetValues` on a DPC the requested valueset is either retrieved from the cache if it exists (a cache hit) or it is retrieved from a web service if it does not (a cache miss). We define *wait time* to be the amount of time that it takes a call to `GetValues` to return a valueset. We define *total wait time* to be the sum of all wait times over the course of a composition run and may be estimated by:

$$W = \sum_{i=1}^m (R_i + N_i + X_i) + \sum_{j=1}^{m+h} L_j \quad (1)$$

where m is the number of cache misses, h is the number of cache hits, R is the average web service response time, N is the average data transfer time, X is average time to parse a response, and L is the cache lookup time. We empirically identified values for N , X and L in our estimation of expected performance.

The effect of the cache is shown in Figure 5 (top). When caching was disabled the wait time increased linearly as the number of components increased (top-left). Enabling the cache within a single composition achieved constant wait time. In the case of distributed compositions there was a super-linear increase in the wait time due to the higher cache lookup time associated with the distributed map. The total data transferred increased linearly with the number of components when the cache was disabled and remained constant at 1.5 GB when the cache was enabled (top-right).

3.2 Prefetching

To evaluate the effect of prefetching we measured the wait time of a single model component linked to a DPC as we varied the web service response time. We used response times that were multiples of the model component's processing time. The DPC prefetched valuesets with a limit of one active web service call at any one time.

In the ideal case there is perfect predictive capability and perfect overlap of the time that the DPC spends prefetching and the time that the model component spends calculating time steps. In this case the total wait time is based on the time spent performing concurrent operations C and performing serial operations S :

$$C = \sum_{i=1}^m (R_i + N_i + X_i) \quad (2)$$

$$S = \sum_{k=1}^t P_k + \sum_{i=1}^{m+h} L_i \quad (3)$$

where t is the total number of time steps performed by the model component and P is the processing time of each time step. The total wait time W with prefetching may be estimated by:

$$W = \max(C - S, 0) + \sum_{i=1}^{m+h} L_i \quad (4)$$

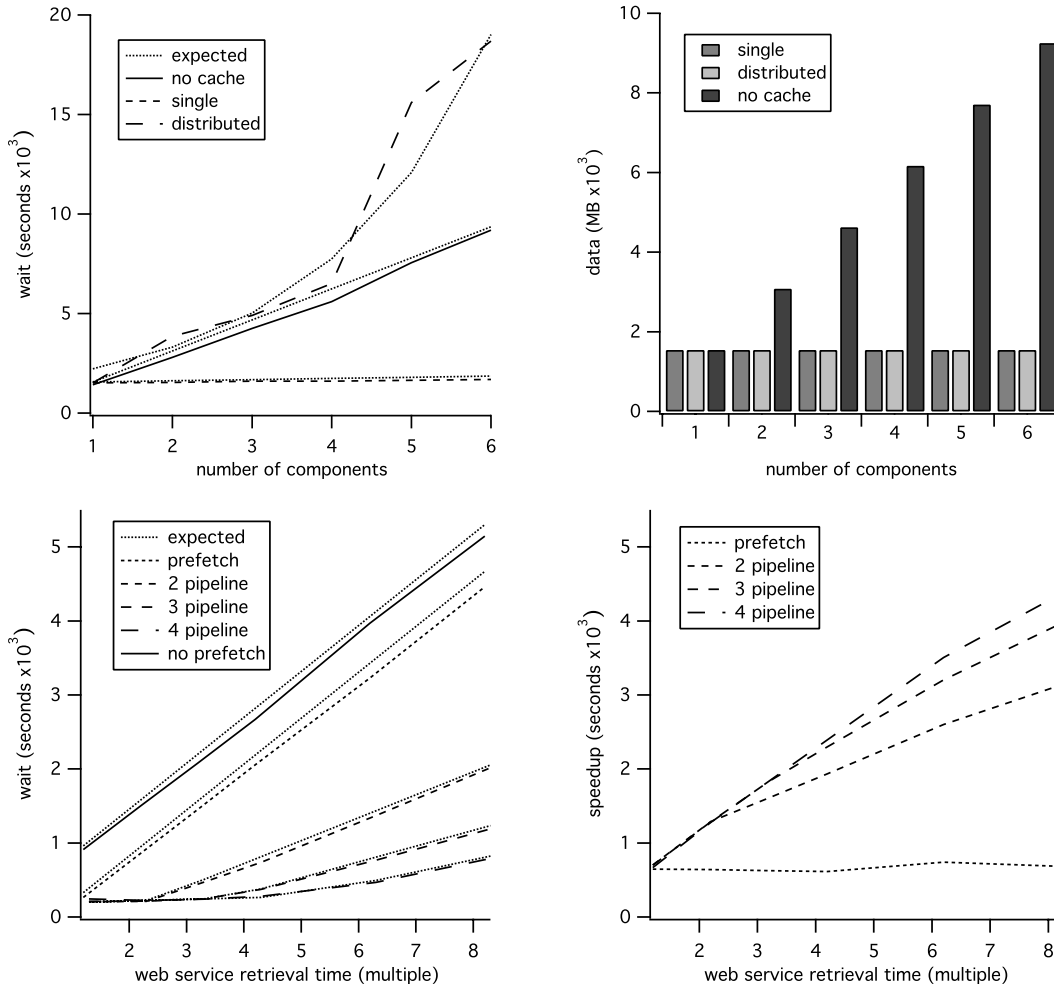


Fig. 5: Performance results: effect of caching (top) and effect of prefetching and pipelining (bottom).

We refer to $R_i + N_i + X_i$ as the *web service retrieval time* as it reflects the total time necessary to retrieve a valueset from a web service.

With prefetching disabled the wait time increased linearly with the web service response time (Figure 5 bottom-left). When prefetching was enabled the total wait time increased at the same rate but was lower by a constant value that corresponded to one web service retrieval time. The speedup in this case was constant (bottom-right).

3.3 Pipelining

To evaluate the effect of multiple concurrent web service calls we measured the wait time of a single model component linked to a DPC as we varied both the number of concurrent requests and the web service response time.

When web service calls are made simultaneously the time spent on concurrent operations is reduced by a factor of the number of simultaneous calls. Thus, the total wait time W

with pipelining may be estimated by:

$$W = \max\left(\frac{C}{Q} - S, 0\right) + \sum_{i=1}^{m+h} L_i \quad (5)$$

where Q is the number of simultaneous web service calls.

The wait time remains constant as long as the number of simultaneous web service calls is the same as the factor by which the web service retrieval time is greater than the model component processing time. The wait time increases sub-linearly with the web service retrieval time once the number of simultaneous web service calls is insufficient to complete all of the necessary prefetching during the model component processing time.

4. Related Work

The synergy between web services and modeling and simulation was recognized quickly as web standards emerged [9]. Web services can provide both a means to access data and to control the execution of online models [9],

[10], [11], [12]. Workflow Management Systems (WMS) provide an infrastructure to setup, execute, and monitor scientific workflows composed of web services [13], [14]. Standard data formats [1], [15] and data access systems [16] have been developed to improve data portability.

There has been recent interest in enabling OpenMI components to interoperate with web services. In one effort [17] a *feature type component* was developed that can retrieve point time series data from a server using the OGC Web Feature Services (WFS) standard. The component steps forward through time and when `GetValues` is invoked it returns the value of the feature type that corresponds to the current time. Another component [18] was developed that can retrieve time series data from a proprietary data platform. To obviate the need for a local data server on the machine that is executing a composition, the component serves as a proxy between the components of a composition and the data server. Our work complements these efforts in the context of computational grids.

5. Conclusions

We presented the design of the Data Provider Component (DPC) for OpenMI components and evaluated its performance. The DPC efficiently retrieves data from multiple web services and delivers the data to OpenMI components that are executing on a cluster. We adapted three common-practice optimizations, caching, prefetching, and pipelining, to the unique behavior and constraints of OpenMI components. General-purpose data components simplify the task of deploying linked models to runtime environments and provide a means for integrating real-time measurement data into their simulations.

The DPC consists of a fetching module and a caching module. The fetching module continuously monitors the data requests made by model components and prefetches data from web services in a pipelined fashion. The caching module services the data requests by extracting the appropriate data from a distributed map that is shared among all DPCs across a cluster.

We evaluated the performance of each of the three optimizations: caching, prefetching, and pipelining. Caching within a single composition achieved linear speedup in wait time as the number of model components increased. Distributed caching was less performant in terms of wait time and would be most advantageous in situations with high latency web services. In both cases the amount of data transferred was minimized and remained constant as the number of model components increased. Prefetching achieved constant speedup in wait time as the web service retrieval time increased and pipelining achieved linear speedup given a sufficient number of simultaneous web service calls.

To mitigate some of the challenges of data management for linked simulations, intelligent, efficient data provider components will become an essential part of any OpenMI

linked model. We believe that this work provides a sound basis for the development of such components.

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] R. K. Rew and G. P. Davis, "The Unidata netCDF: Software for scientific data access," in *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*. Anaheim, California: American Meteorology Society, February 1990, pp. 33–40.
- [2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a common component architecture for high-performance scientific computing," in *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, 1999, p. 13.
- [3] J. B. Gregersen, P. J. A. Gijssbers, and S. J. P. Westen, "OpenMI: Open modeling interface," *J. Hydroinform.*, vol. 9(3), pp. 175–191, 2007.
- [4] P. Taylor, "Harmonising standards for water observation data – discussion paper, OGC 09-124r2," *Open Geospatial Consortium Inc.*, 2010.
- [5] T. Whiteaker, "CUAHSI WaterOneFlow workbook, HIS document 5," *CUAHSI*, 2010.
- [6] I. Zaslavsky, D. Valentine, and T. Whiteaker, "CUAHSI WaterML, OGC 07-041r1," *Open Geospatial Consortium Inc.*, 2007.
- [7] T. Ozturk, "Scalable data structures for java," in *Devoxx*, Metropolis Antwerp Belgium, November 2010.
- [8] The HSQL Development Group, B. Simpson, and F. Toussi, "HyperSQL user guide: HyperSQL database engine (HSQLDB)," *The HSQL Development Group*, 2010.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] H. H. Page, "The HDF Group," 2010, <http://www.hdfgroup.org/HDF5/>.
- [16] 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.
- [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>.

Leveraging Parallelism with CUDA and OpenCL

S. Park¹, D. Shires¹, J. Ross², and D. Richie³

¹U. S. Army Research Laboratory, APG, MD

²High Performance Technologies Inc., Reston, VA

³Brown Deer Technology, Forest Hill, MD

Abstract - Graphics processing units (GPUs), originally designed for computing and manipulating pixels, have become general-purpose processors capable of executing in excess of trillion calculations per second. Taking advantage of GPU's compute power and commodity popularity, the field of computing systems is exhibiting a trend toward heterogeneous platforms consisting of a central processor integrated with graphics hardware. To leverage parallelism within graphics processors, programming approaches employing CUDA and more recent OpenCL framework are evaluated in the context of implementing a ballistic threat field calculation.

Keywords: CUDA, OpenCL, GPGPU

1 Introduction

With the advent of general-purpose computing on graphics processing units (GPGPU), high performance computing has become accessible to general public as the cost of computing has fallen with prevalent and powerful commodity compute resources. Through the combination of consumer market and data-parallel video processing, graphics processing units (GPUs) have become a standard in computer systems and gaining momentum in addressing the processing demands of computationally challenging applications. Ranging from handheld devices to supercomputers, the graphics processing technology is making a mark and contributing to various fields and algorithms [1]. As trends in computing systems are observed, common denominator in Amazon's EC2, Intel's Sandy Bridge, AMD's Fusion, and Apple's A4, is that modern systems are frequently heterogeneous platforms where multiple levels of parallelism must be exploited to utilize the full potential of these systems.

In the ideal case of just looking at maximum hardware capability, theoretical performance of GPU architectures are clearly superior when compared to the

maximum floating-point arithmetic intensity of a CPU. This is evident by the TOP500 list, which ranked Tianhe-1A, the heterogeneous system consisting of CPUs and GPUs, as the world's fastest supercomputer in November 2010 [2]. Higher capability in math processing with graphics processors is achieved by employing a greater compute density in chip design. With more lightweight compute elements, increased arithmetic throughput is achieved. For instance, AMD's high-end products have in excess of a thousand stream processors, leading to a peak performance surpassing a trillion floating-point operations per second (TFLOPs). Starting as accelerators responsible for generating pixels, GPUs are now TFLOP capable devices. With the aid of modern video card technology, a personal high performance computing system can be realized at a fraction of historical supercomputing costs. Moreover, massively parallel structure of graphics processors can be utilized for executing integer operations. The issue rates for various integer and floating-point instructions are documented in [3-5].

There are two leading software APIs for leveraging graphics processors toward general-purpose computing: CUDA and OpenCL. NVIDIA was one of the earlier companies offering software development environment for accessing underlying graphics architecture for parallel computing. As such, a greater adaptation and user base of CUDA exists in GPGPU community. Moreover, this head start translates to more available tools and refined libraries. Similarly, OpenCL addresses the programming of ever increasing parallel devices and is an open standard with an impressive list of participants [6]. One of the distinctions with OpenCL is its ability of targeting wide range devices such as high performance computers, mobile devices, and embedded systems [7]. This paper investigates the programming experiences with the framework of CUDA and OpenCL.

2 CUDA and OpenCL

A cursory look at source codes reveals that CUDA programs tend to be less verbose than OpenCL codes. To support various compute resources, OpenCL implementation requires more setup management of which include querying for platform and device, creating contexts and command queues, building programs, setting kernel arguments, and enqueueing kernels. As a result, OpenCL is able to support various parallel architectures and take advantage of available compute resources within a heterogeneous system. The goal is to define a standard low-level API for parallel computing that provides extreme portability across wide range of computing devices (smartphones to supercomputers).

CUDA uses Open64 compiler for building implementations where as OpenCL rely on LLVM and Clang. A notable feature in OpenCL is the just-in-time compilation support, which builds a kernel program at runtime. This dynamic compilation model consists of two steps: first a code is compiled to a LLVM intermediate representation using the front-end compiler, then the intermediate representation is compiled to a machine code by the back-end compiler. First step is usually done once and the intermediate representation is stored. The application loads the IR and performs the second step at runtime. Installable Client Driver (ICD) allows multiple implementations to co-exist where a code only links to the OpenCL shared library.

3 Factors Affecting Performance

Clear correlation exists between software programming and performance. Understanding of execution model and architectural details of underlying processor is the key in reaching full capacity. Regardless of whether an implementation is in CUDA or OpenCL, knowledge of hardware characteristics and impact on performance is applicable across languages. Programming must reflect architecture specific optimization and make effort to fine tune for a particular hardware. Most likely, a code specialized and optimized specifically for the AMD's Radeon graphics card is not going to perform at the same optimal level when executed on a Nvidia's GPU due to differences in low-level hardware. Typically, CPUs accelerate

through SSE, Nvidia GPUs follow SIMT execution model, and AMD GPUs employ VLIW technique.

Runtime is also tied to how an algorithm is translated and ported to a processing unit. A programmer needs to apply architecture insights during the process of mapping an algorithm to a particular hardware. Understanding memory structure hierarchy, data access behavior, and multithreading operation could result in an order magnitude performance improvement. An excellent implementation is typically obtained via reformulating an algorithm to take advantages of hardware's mode of operation.

3.1 Data Types and Structures

The data type float4 proves helpful to both x86 and AMD GPU architectures, which leads to optimal hardware utilization. The float4 data type translates to packed SSE instructions in x86 processors and maps efficiently to five-way scalar VLIW processors for AMD GPUs [8]. As for NVIDIA products, float4 data type is supported, but the operations involving float4 variables are not included natively (although provided under SDK package). In this project, operation overloading was employed for float4 data types in CUDA to match OpenCL source code. In terms of meeting the requirement for coalesced memory operations, data alignment is automatic for float4 data primitive in CUDA. Having a correct alignment with some unused element access would still result in a single memory transaction.

Data structure arrangement of choice in data-parallel computing is structure of arrays (SoA), which avoids interleaved data pattern apparent in array of structures (AoS). This holds true for CPUs, since SoA structure allows the application to fully utilize the SIMD registers and could lead to reduced memory traffic (only relevant data is loaded into cache). In CUDA, SoA layout satisfies the requirement of the efficient coalesced memory transfer, which in general significantly reduces memory transfer time. In summary, the SoA data structure is a more natural structure for SIMD operations yielding better performance. When impractical to maintain data in SoA format, one option is to restructure AoS to SoA for SIMD computation then stored back in intuitive AoS format.

3.2 Balance of Arithmetic and Memory

For analysis and optimization of multithreaded processors, it is helpful to determine the limiting factors of performance. The goal is to keep the processing elements fully occupied and busy while overlapping the latency of memory accesses. As addressed in [9], optimization on multithreaded hardware model can be counterintuitive. In some cases, increasing arithmetic instructions resulting in lower occupancy can lead to reduced runtime due to extra arithmetic executions masking memory latency. This is an example of insufficient amount of arithmetic that degrades GPU's performance as a result of computing resources in idle state.

During an optimization process, an algorithm of interest should be evaluated for computation and memory limitations. Source code modification to measure memory-only and math-only versions of a kernel is outlined in [10]. This analysis will indicate an entry point and narrow the area of focus for optimization.

4 Ballistic Threat Field Algorithm

The algorithm under investigation is an application that computes ballistic threat field calculations inside a 3D urban terrain modeled as 68,356 triangles. The design of the algorithm is structured such that the development can grow as a ray tracing application. The core calculation computes a line-plane intersection, which generates radiating rays from a shooter's position to each triangle to determine the nearest hit triangle. Once the line of sight calculation is discovered, a hit probability function is applied to calculate the ballistic thread probability. Input data is a triangle file derived from a downloaded file in Google 3D Warehouse represented in XML format. The triangles representing building and terrain surfaces are indexed by a quadtree data structure to take advantage of early termination. The chosen 3D map environment has a balanced mixture of an urban area consisting of building structures and an undeveloped area representing an open rough terrain.

4.1 Runtime Results

Performance results for CUDA and OpenCL was measured and listed in Table 1. The source code was written in languages C, CUDA, and OpenCL. The C version represents a straight-forward single-threaded implementation running on a single core. Because Xeon E5520 has lower clock frequency, C implementation on Xeon 5160 outperforms the quad-core. The CUDA version is only able to target Nvidia GPUs. CUDA implementation performed poorly on the Tesla card where memory transfer seems to be slower on the Tesla cluster configuration. Lastly, the OpenCL version allows for diverse platform execution. For OpenCL source code, the value of local thread size was adjusted according to the targeted hardware. Comparing the runtimes of the sequential and parallelized approaches illustrates the significant impact in runtime based on the programming approach.

Table 1: Execution times

	CUDA	OpenCL	C
GeForce 8800 GTX	1.05 sec	3.22 sec	N/A
Tesla C1060	1.19 sec	0.98 sec	N/A
Firestream 9250	N/A	1.78 sec	N/A
Xeon 5160 Dual-Core 3.0 GHz	N/A	7.67 sec	82.65
Xeon E5520 Quad-Core 2.26 GHz	N/A	1.53 sec	100.76

5 Conclusion

A close examination of processor architectures typically reveals and explains justifications behind hardware features, design choices, and optimization techniques leading to faster processing. Since processor designs are limited by power and heat, many core, VLIW, and SIMD execution models focus on reducing

power consumption while increasing parallelism. In the era of parallel hardware, performance gain is obtained through software programming.

In the end, there is a strong relationship between performance and time investment of applying optimizations. It boils down to learning how architecture affects runtime and programming accordingly to reach maximum hardware utilization. Regardless of programming frameworks and associated languages, the architecture knowledge is portable across software tools in achieving better performance. Algorithms implemented with architecture awareness enable operation to better exploit architecture specifics and features.

6 References

- [1] Nickolls, J., Dally, W.J., "The GPU Computing Era," *Micro, IEEE*, vol.30, no.2, pp.56-69, March-April 2010.
- [2] TOP500. Retrieved February 22, 2011, <http://www.top500.org>
- [3] AMD, "Programming Guide: AMD Accelerated Parallel Processing OpenCL," Jan 2011.
- [4] Nvidia, "NVIDIA CUDA C Programming Guide," Version 3.2, Nov 2010.
- [5] Nvidia, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," V1.1, 2009.
- [6] Khronos Promoting Member Companies. *Khronos Group*. Retrieved February 12, 2011, <http://www.khronos.org/members/promoters>
- [7] A. Munshi, B. Gaster, T. G. Mattson, D. Ginsburg (2010). *OpenCL Programming Guide, Rough Cuts*, Addison-Wesley Professional.
- [8] Behr, D. (2009, September). AMD GPU Architecture. PPAM 2009, Wroclaw, Poland.
- [9] Micikevicius, P. (2010, October). Analysis-Driven Optimization. GTC 2010, San Jose, CA.
- [10] Volkov, V. (2010, October). Better Performance at Lower Occupancy. GTC 2010, San Jose, CA.

Distributed Parallel D8 Up-Slope Area Calculation in Digital Elevation Models

R. Barnes¹, C. Lehman², and D. Mulla³

¹Ecology, Evolution, & Behavior, University of Minnesota, Minneapolis, MN, USA

²College of Biological Sciences, University of Minnesota, Minneapolis, MN, USA

³Soil, Water, & Climate, University of Minnesota, Minneapolis, MN, USA

Abstract—This paper presents a parallel algorithm for calculating the eight-directional (D8) up-slope contributing area in digital elevation models (DEMs). In contrast with previous algorithms, which have potentially unbounded inter-node communications, the algorithm presented here realizes strict bounds on the number of inter-node communications. Those bounds in turn allow D8 attributes to be processed for arbitrarily large DEMs on hardware ranging from average desktops to supercomputers. The algorithm can use the OpenMP and MPI parallel computing models, either in combination or separately. It partitions the DEM between slave nodes, calculates an internal up-slope area by replacing information from other slaves with variables representing unknown quantities, passes the results on to a master node which combines all the slaves' data, and passes information back to each slave, which then computes its final result. In this way each slave's DEM partition is treated as a simple unit in the DEM as a whole and only two communications take place per node.

Keywords: D8, up-slope area, digital elevation model, distributed, parallel, flow accumulation

1. Introduction

DEMs are data structures, usually a rectangular array of floating-point or integer values, representing terrain elevation above some common base level, generally measured via remote sensing techniques or LIDAR.

DEMs are used extensively to model hydrologic processes and properties including soil moisture (based on catchment area), terrain instability (based on slope and catchment area), erosion (based on slope), and stream power (based on slope and catchment area) [5].

Underlying the aforementioned calculations is a flow function responsible for determining what proportion of each cell's flow each neighbouring cell will receive. Perhaps the most widely used function is the D₈ function introduced by O'Callaghan and Mark [3]. This function directs the entirety of each cell's flow to the lowest of its eight surrounding neighbours. This implies that flows *combine but never disperse*: a property we take advantage of. In contrast is the D_∞ method introduced by Tarboton [4], which calculates an angle of steepest descent based on adjacent pairs of

neighbouring cells and directs flow to one or both neighbours along that path.

The accuracy of DEM-based calculations is related to the DEM's resolution. These have gone from thirty-plus meter resolution in the recent past to the sub-meter resolutions becoming available today. Increasing resolution has led to increased data sizes: current data sets are on the order of gigabytes and increasing. While computer processing and memory performance have increased appreciably during this time, legacy equipment and algorithms suited to manipulating smaller DEMs with coarser resolutions make processing these improved data sources costly, if not impossible.

Wallis et al. [5] present one solution for calculating "up-slope area" based on previous work by Mark [2] and O'Callaghan and Mark [3]: a parallel algorithm suitable for both D₈ and D_∞ calculations for use in environments where communication is inexpensive using a queue-based up-slope area function. Although their algorithm, as published, assumes shared memory, this is not a strict requirement.

The algorithm presented here applies only to D₈, is suitable for environments where inter-node communication is expensive relative processing, makes efficient use of multi-processor nodes, and optimises the queue method presented by Wallis et al. [5].

2. Up-Slope Area

Up-slope area A is defined physically for each point p in a watershed as the set of all points P whose flow, if we were to put a liquid in them, would eventually pass through p . Mathematically, this is defined by the recursion relation:

$$A(p) = 1 + \sum_{i=n(p,P)} A(i) \quad (1)$$

Where $n(p, P)$ defines the points neighbouring p , given P . In practice, points are generally represented by cells of some sort, which may or may not be of equal area and may or may not be weighted equally in the calculation. In the case of most DEMs and this paper, the points are represented by square cells with an area and weight of one.

3. The Algorithm

It is assumed that the DEM has been preprocessed to remove pits and flats. Pits are cells with no lower neighbours

or groups of cells from which there is no outlet. In the case of a single cell, the cell is generally raised to the level of its lowest neighbor; in the case of groups, an outlet is usually drilled. Flats are cells with one or more neighbours of equal elevation, these may be resolved by making slight elevation modifications within the flat; Garbrecht and Martz [1] present one approach.

Following this, each of S slave nodes reads in an equal number of horizontal rows of the DEM, with any extra rows allocated to the final slave. In addition to its allotted rows, each slave also reads in two extra rows above and below its strip. This permits the slave to determine flow direction for the cells at the edge of its allotted strip and for the cells bordering it. Henceforth, cells at the edge of a slave's allotted strip are called *edge cells* and are found in either the *TopRow* or the *BottomRow* of the strip.

In this algorithm, the D_8 flow direction of a cell is specified as being towards the neighbour with the steepest slope relative that cell using the Euclidean distance between centers of the two cells. We call the net flow field F .

Throughout this paper procedures marked with a parallel subscript (" \parallel ") may be safely invoked on multiple processors whereas loops marked with a parallel subscript be safely partitioned between processors. Any commands which may lead to race conditions are marked as atomic.

Using the flow field F , each slave finds the dependencies of all its cells using Algorithm 1. A cell c is a dependent of a neighbour n if n 's flow is directed into it; thus, a cell may have 0–8 dependencies, inclusive. If a cell has no dependencies, it is pushed on to the back of a double-ended queue Q ; otherwise, the number of dependencies is stored in an array D .

Algorithm 1 Slaves calculate dependencies

```

1: procedure FINDDEPENDENCIES
Require:  $F, D, Q$ 
2:   for $\parallel$  all  $c$  in  $F$ 
3:     for all  $n$  inputs to  $c$  do
4:        $D(c) \leftarrow D(c) + 1$ 
5:     end for
6:     if  $D(c) = 0$  then
7:       ATOMIC(push  $c$  onto back of  $Q$ )
8:     end if
9:   end for $\parallel$ 
10: end procedure

```

The purpose of the double-ended queue is to minimize contention in multi-processor environments by decoupling the queue's push and pop functions. The **for** \parallel loop on Line 2 may be safely run in parallel because none of the data being read is altered by the function and only one processor will be writing to any given memory location. The queue-push on Line 7 must be done atomically for the algorithm to run safely. It is possible to implement Algorithm 1 such

that there is no contention by having each processor build its own queue and then merging these just after the function completes.

Following Algorithm 1, each slave performs calculates its Internal Up-slope Area A using Algorithm 2. This algorithm begins with the local maxima of the DEM—those cells added to Q by Algorithm 1—which have an up-slope area of one (only themselves) and decrements the dependency counter of the cells they flow into. When a cell's dependency counter reaches zero, it is added to Q . Thus, as the up-slope area of higher-elevation cells becomes known, it is possible to calculate that of lower-elevation cells.

Algorithm 2 Slaves calculate internal up-slope area

```

1: procedure INTERNALUPSLOPE $\parallel$ ( $c$ )
Require:  $F, D, Q, A$ 
2:   if  $c$  was not specified then
3:     ATOMIC( $c \leftarrow$  front of  $Q$ )
4:     if  $c$  was not set then
5:       return
6:     end if
7:   end if
8:    $A(c) \leftarrow 1$ 
9:   for all  $n$  inputs to  $c$  do
10:     $A(c) \leftarrow A(c) + A(n)$ 
11:   end for
12:    $n \leftarrow$  downslope neighbor of  $c$ 
13:   if  $n$  exists then
14:      $D(n) \leftarrow D(n) - 1$ 
15:     if  $D(n) \neq 0$  then
16:        $n \leftarrow$  NULL
17:     end if
18:   end if
19:   return INTERNALUPSLOPE( $n$ )
20: end procedure

```

Algorithm 2 is defined using tail recursion and, to avoid excessive stack sizes, should be implemented appropriately. Since Q cannot be equitably divided among processors beforehand, it is necessary for each processor to atomically pop cells from it when necessary. However, the depth-first search embodied by the recursion on Line 19 reduces contention on Q .

At this point, information from other slaves is required to resolve the remaining dependencies. Figure 1 depicts this conceptually. **A** represents a flow path originating in the current slave's portion of the DEM: all its dependencies can be satisfied with the information available to the slave, but these results must be communicated to neighbouring slaves.

If we assume that flow is generally directed to the lower-right, then **B** represents an up-slope area originating in a different slave and terminating in this one. In such a case a

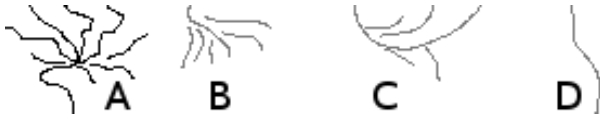


Fig. 1

CONCEPTUAL IMAGE OF DEPENDENCIES AND RESOLUTIONS

single communication between the slaves provides sufficient information to compute up-slope area.

C represents a similar situation, but two communications are necessary because the flow path loops back up: one to resolve this slave's dependencies and another to resolve its neighbour's.

D represents a situation in which two communications are necessary to pass information from one neighbouring slave to another through the current slave.

If **C** and **D** link through the upper neighbouring slave, then four communications would be necessary. Or, if flow is directed to the upper-left, and **B**, **C**, and **D** are linked, then five communications would be necessary. We denote the average number of communications a slave will need to make as its "winding factor" ϕ . In situations where communication is cheap, ϕ is unimportant and algorithms such as that presented by Wallis et al. [5] are effective. In situations where communication is expensive, a different algorithm, such as that presented here, is necessary.

If each slave denotes external inputs from other slaves by variables, it is possible to continue the calculation with minimally inter-nod communication.

We call edge cells with unresolved dependencies in other slaves' DEMs *receivers*. They receive information from at most 3 of the other slave's edge cells. Cells upon which another slave's DEM depends are either *givers* or *joiners*. A giver cell's upslope-area has already been calculated by Algorithm 2; a joiner is the dependent of one or more receiver cells.

It is a property of the D_8 flow function that flows join but never disperse and, therefore, that every joiner is a dependent of *at least* one receiver while each receiver ultimately feeds into *at most* one joiner. This property does not hold for the D_∞ flow function, preventing its use in this context.

Algorithm 3 presents one possible way of preparing the slave to perform the External Up-slope Area calculation. The original dependency grid is saved on Line 2 to be used later in calculating the true up-slope area. Once this is done, the external dependencies of all edge cells are removed and those cells which are "satisfied" are added to Q . The cell is marked as a receiver by assigning it a special variable name on Line 5.

$Cell_V$, first mentioned on Line 5, is a map between cells c and variable names, which are represented as globally-unique numbers with -1 acting as a special value used to

denote receivers. Algorithm 3 marks all cells as receivers; Algorithm 4 will remark those which are not.

Algorithm 3 Slaves prepare to calculate external up-slope area

```

1: procedure SATISFYRECEIVERS
2:   Require:  $F, D, D_O, Cell_V, Q$ 
3:    $D_O \leftarrow D$ 
4:   for|| all  $c$  in  $TopRow$ 
5:      $V \leftarrow NEWVARIABLE$ 
6:      $Cell_V(c) \leftarrow -1$ 
7:     for all  $n$  above-inputs to  $c$  do
8:        $D(c) \leftarrow D(c) - 1$ 
9:     end for
10:    if  $D(c) = 0$  then
11:      ATOMIC(push  $c$  onto back of  $Q$ )
12:    end if
13:  end for||
14:   $\triangleright$  Analogous code for  $BottomRow$ 
15: end procedure

```

Algorithm 4 calculates External Up-slope Area, keeping track of how cells depend on receivers. When a cell is popped off Q , it is associated with a new variable name V (Line 7) which is unique across all slaves and used in future iterations to keep track of the flow path's origin using the multimap¹ $Origin$ (Line 8). Since only edge cells are in Q , new variables are only formed at the edge cells.

After incrementing a cell c 's up-slope area, the algorithm attempts to follow the flow path from c to its neighbour n (Line 19). If there is a neighbour and its dependencies are satisfied, the algorithm recurses, maintaining knowledge of which edge cell its flow path originated at using V (Line 23). If n 's dependencies are not satisfied, the algorithm uses the map $Cell_V$ to inform a future iteration of the existence of the flow path it's about to abandon (Line 26).

On Lines 14–18, after the cell's up-slope area has been incremented, the algorithm inspects the present cell and cells flowing into it to see if it or they are part of a previously abandoned flow path. If so, it merges that flow path with the present one and continues.

Ultimately, the algorithm reaches a point where no down-slope neighbour exists and there are no more cells in Q . It abandons the flow path (Line 26), recurses, and exits (Line 5). The result of this algorithm is a map $Cell_V$ indexed on the joiner cells. These cells are part of the border of the current slave and the edge of the adjacent slave—they are the adjacent slave's receivers. So $Cell_V$, coupled with $Origin$ link one slave's receivers to another's.

¹A hash table which associates the same key with multiple values. In this paper empty or nonexistent keys in maps and multimaps always return NULL when their values are cell identifiers and 0 when their values are used mathematically.

Algorithm 4 Slaves calculate external up-slope area

```

1: procedure EXTERNALUPSLOPE||( $V, c$ )
Require:  $F, D, Q, A, Cell_V, Origin$ 
2:   if  $c$  was not specified then
3:     ATOMIC( $c \leftarrow$  front of  $Q$ )
4:     if  $c$  was not set then
5:       return
6:     end if
7:      $V \leftarrow$  NEWVARIABLE
8:      $Origin(V) \leftarrow c$ 
9:   end if

10:   $A(c) \leftarrow 1$ 
11:  for all  $n$  inputs to  $c$  do
12:     $A(c) \leftarrow A(c) + A(n)$ 
13:  end for

14:  for all  $n$  inputs to  $c$  and  $c$  itself do
15:    append  $Origin(Cell_V(n))$  to  $Origin(V)$ 
16:    erase  $Origin(Cell_V(n))$ 
17:    erase  $Cell_V(n)$ 
18:  end for

19:   $n \leftarrow$  downslope neighbor of  $c$ 
20:  if  $n$  exists then
21:     $D(n) \leftarrow D(n) - 1$ 
22:    if  $D(n) = 0$  then
23:      return EXTERNALUPSLOPE( $V, n$ )
24:    end if
25:  end if
26:   $Cell_V(c) = V$ 
27:  return EXTERNALUPSLOPE( $-$ ,  $-$ )
28: end procedure

```

Each slave now sends $Cell_V$, $Origin$, and the up-slope area of its bordering cells to the master node.

Conceptually, the situation is exemplified by Figure 2. Data from different slaves is separated by wide vertical gaps; it is important to remember that the cells bordering these gaps, though they appear in different slaves, are, in fact, the same in the DEM. Within a slave's data, receivers and joiners form sometimes-complicated linkages while between slaves the linkages are simple; from the master's perspective, the slaves combine to form a directed acyclic graph.

In Figure 2, receivers are marked by variables. These are passed along to joiners, accumulating up-slope area along the way. Therefore, the joiners are represented by the sum of their associated receivers' variables and the upslope area connecting them to each receiver. Givers are represented by a pure number: this is a true up-slope area and a starting point for the next calculation.

Algorithm 5 prepares the slaves' data for processing by locating giver cells and determining how many dependencies

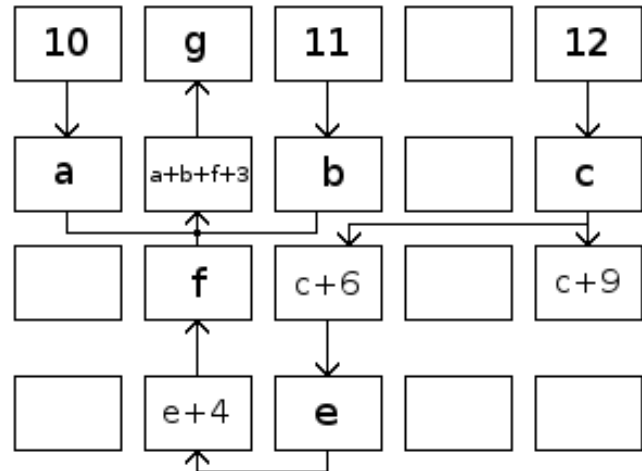


Fig. 2

CONCEPTUAL IMAGE OF MASTER NODE'S DATA

each joiner has. Since the map keys are globally unique, all the slaves' $Cell_V$ s are combined; likewise, the $Origins$. The function *Invert* on Line 14 turns $Origin$'s keys into values and vice versa. The result is a simple map serving the same purpose as the flow field array F of Algorithm 2. The dependency array holds: 1 for receivers since they only depend on data from cells in adjacent slaves; -1 for givers so that they are not later mistaken for receivers; and a positive number, 1 or more, for joiners.

Algorithm 5 Master node prepares received data

```

1: procedure MASTERPREP
Require:  $D, Q, Cell_V, Origin$ 
2:   for all Slaves  $s$ 
3:     for all  $c$  in ( $TopRow, BottomRow$ ) do
4:       if  $Cell_V(c)$  is undefined then  $\triangleright$  Giver
5:         ATOMIC(push  $c$  onto back of  $Q$ )
6:          $D(c) \leftarrow -1$ 
7:       else if  $Cell_V(c) = -1$  then  $\triangleright$  Receiver
8:          $D(c) \leftarrow 1$ 
9:       else  $\triangleright$  Joiner
10:         $D(c) \leftarrow$  LENGTH( $Origin(Cell_V(C))$ )
11:      end if
12:    end for
13:  end for
14:   $Destination \leftarrow$  INVERT( $Origin$ )
15: end procedure

```

Algorithm 6 solves the system of equations presented by the slaves using the same methodology as in Algorithms 2 and 4. Lines 8–18 identify the cell's destination. If one has not been explicitly declared the cell must either be a

giver or a joiner and its destination is implied as being to the slave above/below it, provided that cell is a receiver. Figure 2 depicts one situation wherein this would arise. The destination of the cell in the bottom right labeled “ $c + 9$ ” is implied to be directly below it; however, the two are unconnected. If the destination is valid, the algorithm continues in the usual fashion.

Algorithm 6 Master node calculates up-slope area

```

1: procedure MASTERUPSLOPE||( $c$ )
Require:  $A, D, Q, Cell_V, Destination$ 
2:   if  $c$  was not specified then
3:     ATOMIC( $c \leftarrow$  front of  $Q$ )
4:     if  $c$  was not set then
5:       return
6:     end if
7:   end if

8:    $n \leftarrow Destination(c)$ 
9:   if  $n$  is undefined then
10:    if  $c$  is a top cell then
11:       $n \leftarrow$  cell above  $c$ 
12:    else
13:       $n \leftarrow$  cell below  $c$ 
14:    end if
15:    if  $Cell_V(n) \neq -1$  then      ▷ Not A Receiver
16:       $n \leftarrow$  NULL
17:    end if
18:  end if
19:  if  $n$  is undefined then
20:    return MASTERUPSLOPE(-)
21:  end if

22:   $A(n) \leftarrow A(n) + A(c)$ 
23:   $D(n) \leftarrow D(n) - 1$ 
24:  if  $D(n) \neq 0$  then
25:     $n \leftarrow$  NULL
26:  end if
27:  return MASTERUPSLOPE( $n$ )
28: end procedure

```

Once Algorithm 6 is completed, the area of each slaves’ top and bottom rows are returned. Since Algorithm 4 set the areas of the slaves’ receivers to one and calculated up-slope area that, we simply have to add each receivers’ incoming area to all its dependents. Line 9 of Algorithm 7 enables this by setting up a map between receivers and their incoming variables. This map will later be used to keep track of which incoming areas belong to which flow path. Finally, the slaves run Algorithm 8.

Algorithm 8 is similar to Algorithm 4 insofar as it propagates variables forward. However, rather than propagating variables and combining flow paths, only the incoming areas are propagated forward.

Algorithm 7 Slaves prepare incoming data

```

1: procedure PREPFINALISEINTERNAL
Require:  $F, D, D_O, Cell_V, Q$ 
2:    $D \leftarrow D_O$ 
3:   for|| all  $c$  in  $TopRow$ 
4:     for all  $n$  above-inputs to  $c$  do
5:        $D(c) \leftarrow D(c) - 1$ 
6:     end for
7:     if  $D(c) = 0$  then
8:       ATOMIC(push  $c$  onto back of  $Q$ )
9:        $Area_D(c) \leftarrow A_{incoming}(c)$ 
10:    end if
11:  end for||
12:  ▷ Analogous code for  $BottomRow$ 
13: end procedure

```

Algorithm 8 Slaves finalise internal up-slope areas

```

1: procedure FINALISEINTERNAL||( $S, c$ )
Require:  $F, D, Q, A$ 
2:   if  $c$  was not specified then
3:     ATOMIC( $c \leftarrow$  front of  $Q$ )
4:     if  $c$  was not set then
5:       return
6:     end if
7:      $S \leftarrow Area_D(c)$ 
8:   end if

9:   for all  $n$  inputs to  $c$  do
10:     $S \leftarrow S + Area_D(c)$ 
11:    erase  $Area_D(c)$ 
12:  end for
13:   $A(c) \leftarrow A(c) + S$ 

14:   $n \leftarrow$  downslope neighbor of  $c$ 
15:  if  $n$  exists then
16:     $D(n) \leftarrow D(n) - 1$ 
17:    if  $D(n) = 0$  then
18:      return FINALISEINTERNAL( $S, n$ )
19:    end if
20:  end if
21:   $Area_D(c) = S$ 
22:  return FINALISEINTERNAL(-, -)
23: end procedure

```

4. Conclusions

The algorithm presented here makes efficient use of multi-processor nodes and is well-suited to environments where communication is expensive and must be kept to a minimum. Each slave communicates with a master node only once and the master node communicates with each slave only once. In the case of a single node performing the calculations, it would be possible to store intermediate results to disk or layer this algorithm, allowing the node to process arbitrarily large DEMs dependent only on disk space.

References

- [1] Jurgen Garbrecht and Lawrence W Martz. The assignment of drainage direction over flat surfaces in raster digital elevation models. *Journal of Hydrology*, 193:204–213, June 1997. ISSN 00221694. doi: 10.1016/S0022-1694(96)03138-1. URL <http://linkinghub.elsevier.com/retrieve/pii/S0022169496031381>.
- [2] D.M. Mark. *Modelling in Geomorphological Systems*, chapter Network models in geomorphology, pages 73–97. John Wiley & Sons, 1988.
- [3] John O'Callaghan and David Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics, and Image Processing*, 28(3):323–344, December 1984. ISSN 0734189X. doi: 10.1016/S0734-189X(84)80011-0. URL <http://linkinghub.elsevier.com/retrieve/pii/S0734189X84800110>.
- [4] David G Tarboton. A New Method For Determining Flow Directions And Upslope Areas In Grid Digital Elevation Models. *Water Resources Research*, 33(2):309–319, 1997.
- [5] Chase Wallis, Dan Watson, David Tarboton, and Robert Wallace. Parallel Flow-Direction and Contributing Area Calculation for Hydrology Analysis in Digital Elevation Models. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1–5, Las Vegas, Nevada, USA, 2009.

Selecting the Best Tridiagonal System Solver Projected on Multi-Core CPU and GPU Platforms

Pablo Quesada-Barriuso¹, Julián Lamas-Rodríguez¹, Dora B. Heras¹, Montserrat Bóo¹, Francisco Argüello¹

¹Centro de Investigación en TecnoloXías da Información (CITIUS), Univ. of Santiago de Compostela, Spain

Abstract—*Nowadays multicore processors and graphics cards are commodity hardware that can be found in personal computers. Both CPU and GPU are capable of performing high-end computations. In this paper we present and compare parallel implementations of two tridiagonal system solvers. We analyze the cyclic reduction method, as an example of fine-grained parallelism, and Bondeli's algorithm, as a coarse-grained example of parallelism. Both algorithms are implemented for GPU architectures using CUDA and multi-core CPU with shared memory architectures using OpenMP. The results are compared in terms of execution time, speedup, and GFLOPS. For a large system of equations, 2²², the best results were obtained for Bondeli's algorithm (speedup 1.55x and 0.84 GFLOPS) for multi-core CPU platforms while the cyclic reduction (speedup 17.06x and 5.09 GFLOPS) was the best for the case of GPU platforms.*

Keywords: Tridiagonal system solver, multi-core CPU, OpenMP, GPU, CUDA

1. Introduction

The tridiagonal system solvers are widely used in the field of scientific computation, especially in physical simulations. The tridiagonal matrix algorithm, also known as the Thomas Algorithm [1], is one of the most known algorithms used to solve tridiagonal systems. It is based on the Gaussian elimination [2] to solve dominant diagonal systems of equations. As it is not suitable for parallel implementations, new algorithms such as cyclic reduction [3], [4] or recursive doubling [5] were developed to exploit fine-grained parallelism. Other parallel tridiagonal solvers like [6], [7], [8] were designed with a coarse-grained parallelism in mind.

OpenMP is designed for shared memory architectures [9]. The development in recent years of cluster-like architectures favoured the Multiple Instructions Multiple Data programming languages, particularly MPI. However, recent multi-core architectures have increased interest in OpenMP which has led us to choose this API for our CPU implementations.

The Computed Unified Device Architecture [10] developed by NVIDIA provides support for general-purpose computing on graphics hardware [11] with a fine-grained and coarse-grained data parallelism. However, in most cases, neither sequential nor parallel algorithms can be implemented directly into the GPU. The algorithm's design needs to be adapted in order to exploit each architecture.

In this paper we present and compare a parallel implementation of cyclic reduction as described in [12] and one of Bondeli's algorithm [8]. Both algorithms are valid for large systems of equations. They have been implemented for multi-core CPUs using OpenMP and for GPU computing using CUDA. The data access patterns and workflows of the algorithms are divided into several stages, making them good candidates for a parallel implementation using OpenMP, but a challenge for a GPU version owing to the specific features of this architecture. We shall analyze the performance of the cyclic reduction given that it was one of the first parallel algorithms described for solving tridiagonal systems. The cyclic reduction algorithm was focused towards fine-grained parallelism which could be achieved into the GPU architecture. And we shall analyze Bondeli's algorithm as an example of coarse-grained parallelism in a divide-and-conquer fashion, more suited to a multi-core CPU architecture.

In Section 2 we briefly present the proposed algorithms. An overview of the OpenMP programming model and the implementation of the algorithms are presented in Section 3. In Section 4 we present the GPU Architecture and CUDA programming model with the implementations of the algorithms. The results obtained for each proposal are discussed in Section 5. Finally, the conclusions are presented in Section 6.

2. Tridiagonal System Algorithms

The objective is solving a system of n linear equations $A\vec{x} = \vec{d}$, where A is a tridiagonal matrix, i.e.:

$$\begin{bmatrix} b_1 & c_1 & & 0 \\ a_2 & b_2 & \ddots & \\ & \ddots & \ddots & c_{n-1} \\ 0 & & a_n & b_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix}. \quad (1)$$

We shall denote each equation of this system as:

$$E_i \equiv a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \quad (2)$$

for $i = 1, \dots, n$ where $x_0 = x_{n+1} = 0$.

In the following subsections, the tridiagonal system solvers of cyclic reduction and Bondeli's algorithm are presented.

2.1 Cyclic reduction

The cyclic reduction algorithm [12] starts with a *forward reduction* stage where the system is reduced until a unique

Figure 2 shows each step of Bondeli's algorithm for an 8-equation system divided into 2 blocks of size 4. Each square represents an Equation (6) and (7) in steps 1 and 2 respectively, and the computation of an unknown in step 3. In this example $3p - 2 = 4$ subsystems are solved in the first step and an intermediate tridiagonal system of size $2p - 2 = 2$ is built and solved in the second step. Note that Equations (6d) and (6e) are not generated in this example due to the small number of blocks of the first step.

3. Tridiagonal System Solvers Implementations with OpenMP

In this section we present an overview of the OpenMP programming model and the parallel implementations of the two algorithms introduced in Section 2.

3.1 OpenMP Overview

OpenMP is the standard Application Program Interface (API) for parallel programming on shared memory architectures [9]. Communication and coordination between threads is expressed through read/write instructions of shared variables and other synchronization mechanisms. It comprises compiler directives, library routines and environment variables and is based on a fork-join model (see Figure 3) where a master thread creates a team of threads that work together on single program multiple data (SPMD) constructs.

In shared memory architectures OpenMP threads access the same global memory where data can be *shared* among them or exclusive (*private*) for each one. From a programming perspective, data transfer for each thread is transparent and synchronization is mostly implicit. When a thread enters a parallel region it becomes the master, creates a thread team and forks the execution of the code among the threads and itself. At the end of the parallel region the threads join and the master resumes the execution of the sequential code.

Different types of worksharing constructs can be used in order to share the work of a parallel region among the threads [13]. The *loop construct* distributes the iterations of one or more nested loops into *chunks*, among the threads in the team. By default, there is an implicit barrier at the end of a loop construct. The way the iterations are split depends on the schedule used in the loop construct [13]. On the other hand, the *single construct* assigns the work on only one of the threads in the team. The remaining threads wait until the end of the single construct owing to an implicit barrier. This type of construct is also known as non-iterative worksharing construct. Other types of worksharing constructs are available.

Although there are implicit communications between threads through access to shared variables or the implicit synchronization at the end of parallel regions and worksharing constructs, explicit synchronization mechanisms for mutual exclusion are also available in OpenMP. These are critical or atomic directives, lock routines, and event synchronization directives.

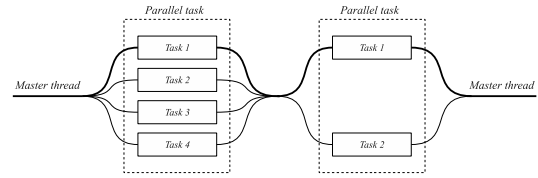


Fig. 3: OpenMP fork-join model.

OpenMP is not responsible for the management of the memory hierarchy but certain issues regarding cache memory management should be borne in mind. There are two factors that determine whether a loop schedule is efficient: data locality and workload balancing among iterations. The best schedule that we can choose when there are data locality and a good workload balance is static with a chunk size of $q = n/p$, where n is the number of iterations and p the number of threads. In other cases dynamic or guided schedules may be adequate.

When a cache line, shared among different processors, is invalidated as a consequence of different processors writing in different locations of the line, false sharing occurs. False sharing must be avoided as it decreases performance due to cache trashing. One way to avoid false sharing is to divide the data that will be accessed by different processors into pieces of size multiple of the cache line size. A good practice to improve cache performance is to choose a schedule with a chunk size that minimizes the requests of new chunks and that is multiple of a cache line size.

3.2 OpenMP implementations

In this section we present our OpenMP proposals, for the implementation of the tridiagonal system solvers: cyclic reduction and Bondeli's algorithm.

3.2.1 Cyclic reduction

For the two stages of the cyclic reduction algorithm we distribute $q = n/p$ equations among the threads, where n is the

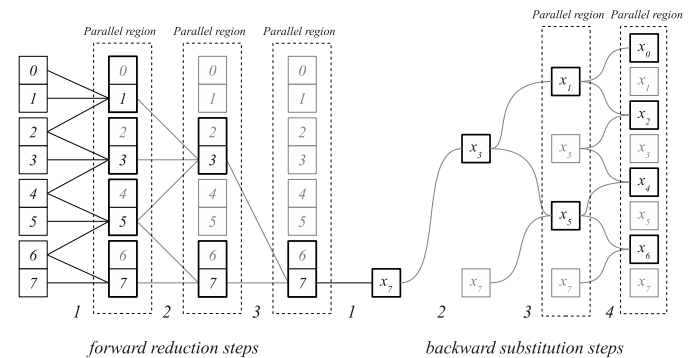


Fig. 4: Work shared among the threads in the forward reduction and backward substitution stage of cyclic reduction for an 8-equation system.

number of equations and p the number of threads. The storage system of this implementation employs five arrays: three matrix diagonals, right-hand side, and unknowns. All arrays are shared among the threads to satisfy the data dependencies.

Figure 4 (left) shows the access pattern for each thread in the forward reduction stage for the case of an 8-equation system. Each thread reduces q consecutive equations (2 equations in Figure 4) except for the last $\log_2 p$ steps, where p is the number of threads, as there are more threads than there are equations to reduce. An implicit barrier at the end of each step keeps the reduction stage synchronized. Figure 4 (right) shows the access pattern for each thread in the backward substitution stage, where the work is shared among the threads in the same way as in the reduction stage.

3.2.2 Bondeli's algorithm

For this implementation the different steps that form Bondeli's algorithm, described in Section 2.2, are executed in the same parallel region saving the time needed to create new thread teams when entering a new parallel region. We use implicit barriers to synchronize the different steps. The storage system of these implementations consists of five arrays that store the three diagonals, the right-hand coefficients and the unknowns of the tridiagonal system. The values of vectors \vec{y}_i and \vec{z}_j computed in the first step are stored in two separate arrays, and the intermediate system $\mathcal{H}\vec{\alpha} = \vec{\beta}$ described in the second step is stored in a similar way to the original system; i.e. in five arrays. All arrays are shared among the threads in order to read and write the data needed in each step.

The tridiagonal system is divided into blocks of size $k \times k$ where $k = n/p$, where n is the number of equations and $3p-2$ the number of threads. The resulting subsystems are solved in the first step of the algorithm. This is the most costly step where

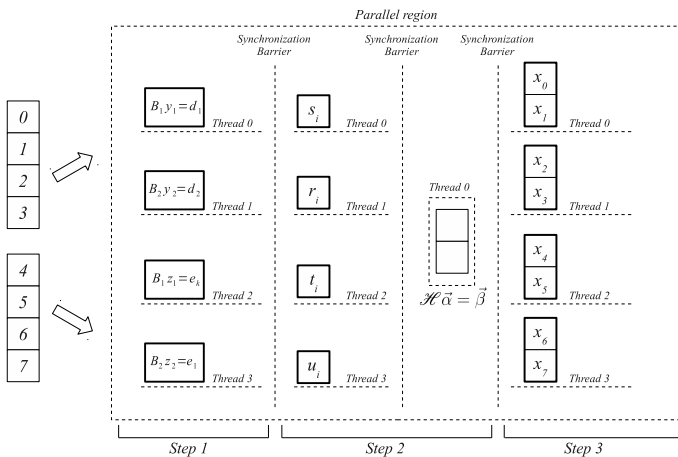


Fig. 5: Work shared among the threads in the different steps of the implementation of Bondeli's algorithm for an 8-equation system.

the subsystems are solved using the Thomas algorithm [1].

Figure 5 shows the worksharing constructs in the different steps of Bondeli's algorithm for an 8-equation system divided into 2 blocks of size 4. The work shared among the threads in this implementation for each step of the algorithm is as follows:

- 1) First step.- Solve the linear tridiagonal systems of size $k = n/p$. Each thread requests one subsystem of equations $B_i \vec{y}_i = \vec{d}_i$, one $B_1 \vec{z}_1 = \vec{e}_k$ or $B_p \vec{z}_{2p-2} = \vec{e}_1$, Equations (6a)-(6c), Section 2.2. In our implementation the original tridiagonal system is split into 2 blocks, so Equations (6d)-(6e) do not need to be solved. A barrier at the end of the last directive synchronizes all the threads before the next step.
- 2) Second step.- Compute α_i by solving the tridiagonal system $\mathcal{H}\vec{\alpha} = \vec{\beta}$. The elements s_i, r_i, t_i of \mathcal{H} and u_i of $\vec{\beta}$ (see [8] for details) are calculated in parallel by the different threads. The system $\mathcal{H}\vec{\alpha} = \vec{\beta}$ of $2p-2$ equations (2 equations in Figure 5) is solved by one thread. The thread team is synchronized before the next step.
- 3) Third step.- Compute the solution with two SAXPY-operations. The final solution is computed sharing $q = n/t$ equations among the threads, where n represents the number of equations and t the number of threads.

4. Tridiagonal System Solver Implementations with CUDA

In this section we present an overview of the GPU architecture and the CUDA implementations of the proposed algorithms introduced in Section 2.

4.1 GPU Overview

CUDA technology is the Compute Unified Device Architecture for NVIDIA programmable Graphic Processor Units [10]. This architecture is organized into a set of streaming multiprocessors (SMs) each one with many-cores or streaming processors (SPs). These cores can manage hundreds of threads in a Simple Program Multiple Data (SPMD) programming model. The number of cores per SM depends on the device architecture [10], i.e. the NVIDIA's G80 series has a total of 128 cores in 16 SMs each one with 8 SPs. Figure 6 shows a schema of a CUDA-capable GPU.

A CUDA program, which is called a kernel, is executed by thousands of threads grouped into blocks. The blocks are arranged into a grid and scheduled in any of the available cores enabling automatic scalability for future architectures. If there are not enough processing units for the blocks assigned to a SM, the code is executed sequentially. The smallest number of threads that are scheduled is known as a warp. All the threads within a warp execute the same instruction at the same time. The size of a warp is implementation defined and it is related to shared memory organization, data access patterns and data flow control [10], [14].

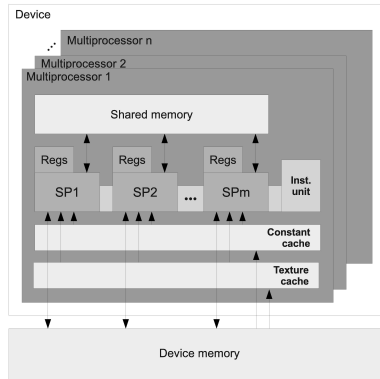


Fig. 6: A schema of streaming multiprocessors in a CUDA-capable GPU.

The memory hierarchy is organized into a *global memory* and a read-only *constant and texture memories*, with special features such as caching or prefetching data. These memories are available for all the threads. There is an on-chip *shared memory* space available per block enabling extremely fast read/write access to data but with the lifetime of the block. Finally, each thread has its own local and private memory.

There are mechanisms to synchronize threads within a block but not among different blocks. Due to this restriction data cannot be shared among blocks. This becomes a challenge when a thread needs data which have been generated outside its block.

4.2 CUDA implementations

In this section we present our proposals, using CUDA, for the implementation of the tridiagonal solvers under analysis.

The major challenge in pursuing an efficient implementation of a tridiagonal system solver in the GPU is the distribution of the system among the thread blocks. For small systems, the solution of the entire system can be computed using only shared memory and the register space without writing the intermediate results back to global memory. This can result in substantial performance improvements, as can be observed in [15], [16].

In the case of very large tridiagonal systems, the equations must be distributed among thread blocks, and a mechanism must be implemented to combine the partial operations performed by these individual blocks and to obtain the global solution. In standard FFT approaches, which present similar access patterns, the transformation of a large sequence can be computed by combining FFTs of subsequences that are small enough to be handled in shared memory. Data are usually arranged into a bidimensional array, and the FFTs of the subsequences are computed along rows and columns [17], [18]. This technique cannot be applied directly to tridiagonal systems solvers, since the factorization of a tridiagonal system into subproblems to be independently computed is not as regular as the FFT case.

4.2.1 Cyclic reduction

When the cyclic reduction method is implemented in global memory, the forward reduction stage can be implemented through a loop executed by the host, which calls a CUDA kernel at each step. The forward reduction kernel call invokes one thread for each equation E_i of the system that is modified in the current step and executes the operations stated in Expression (3). In this stage, the number of invoked threads drops by a factor of two at each new step. Once the forward reduction is completed, the backward substitution begins, and a second kernel is called, where an unknown x_i is assigned to each thread which computes the value thereof by applying Expression (4). At each of the second kernel calls the number of threads increases by a factor of two.

In our GPU implementation of the cyclic reduction, we used the shared memory space as much as possible [19]. The forward reduction stage is done in two kernels. A first kernel splits the equations into several non overlapping blocks. Each thread within a block copies an equation from global memory into the shared memory of its block. This kernel then solves as many steps of the reduction stage as possible with the data stored in the shared memory. Finally the intermediate results are copied back to global memory. The second kernel computes the equations that can not be solved due to data dependencies with the adjacent blocks. These equations are computed directly into global memory what is more efficient in this case due to the small number of equations remaining. Both kernels are successively executed until the forward reduction stage is completed. The backward substitution stage can be computed in a similar way.

4.2.2 Bondeli's algorithm

In this implementation, the different stages of Bondeli's algorithm, described in Section 2.2, are executed in several kernels because of the global synchronization needed in each stage [19]. The original tridiagonal system is split in small subsystems which can be solved independently. All subsystems are solved in just one kernel call, and the solution is nearly entirely performed in the shared memory space. This is possible due to the small size of the subsystems, which can now be assigned to different blocks of threads and be solved independently in parallel using the cyclic reduction algorithm.

5. Results

We have evaluated our proposals on a PC with an Intel Core 2 Quad Q9450 with four cores at 2.66 GHz and 4 GB of RAM. The main characteristics of the memory hierarchy are described in [20]. Each core has a L1 cache divided into 32 KB for instructions and 32 KB for data. With respect to L2, it is an unified cache of 6 MB shared by 2 cores (12 MB in total). Cache lines for the L1 and L2 caches are 64 bytes wide. The code has been compiled using gcc version 4.4.1 with OpenMP 3.0 support under Linux. For the CUDA

implementations we ran the algorithms on a NVIDIA GeForce GTX 295. The CUDA code has been compiled using nvcc also under Linux. In both cases the code was compiled without optimizations.

The results are expressed in terms of execution times, speedups and GFLOPS. The execution times were obtained as an average of one hundred executions. The speedup are calculated for both OpenMP and CUDA implementations with respect to the sequential implementation of the Thomas algorithm. The execution times for the OpenMP implementations are those corresponding to 4 threads. These times include the creation of the thread team which is OpenMP implementation defined. As the final goal is to execute the tridiagonal system solvers within a larger computational algorithm, the data transfer between the host and the device in the CUDA implementations is not included in the execution times.

5.1 OpenMP

Figure 7 shows the results for the OpenMP implementations of the cyclic reduction and Bondeli's algorithm for increasing system sizes from 2^{10} to 2^{22} in steps of 2^2 . The best results for the cyclic reduction implementation are obtained for 2^{16} -equation systems, which presents a speedup of 2.15x. The decrease in speedup for larger systems is due to the storage requirements of the algorithm. For example, a system of 2^{20} equations requires 20 MB of memory which is more than the 12 MB of the L2 cache. Consequently, the number of cache misses increases and, given that the data accesses do not present spatial locality, the execution time also increases.

For small systems Bondeli's algorithm presents lower speedups than the cyclic reduction algorithm, mainly due to the high arithmetic requirements of the algorithm. Nevertheless, its speedup curve presents a more linear behavior than for the cyclic reduction algorithm because the intermediate subsystems are solved with the Thomas algorithm, which presents a sequential access data pattern. Another reason is a good workload

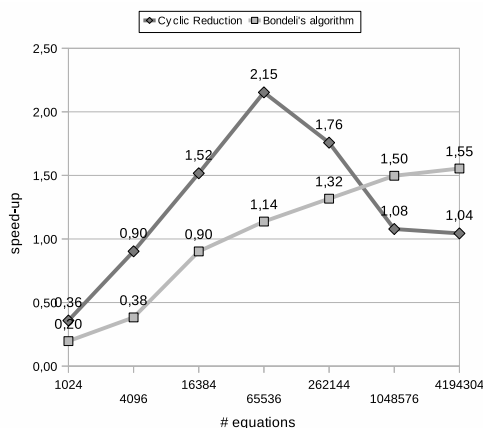


Fig. 7: OpenMP speedup of cyclic reduction and Bondeli's algorithm with different tridiagonal system sizes.

Table 1: Performance results obtained for a 2^{20} -equation system (CR denotes cyclic reduction).

Algorithm	Thomas	CR	Bondeli
Sequential			
Time	0.0547 s	—	—
GFLOPS	0.14		
OpenMP			
Time	—	0.0507 s	0.0365 s
GFLOPS		0.44	0.78
CUDA			
Time	—	0.0029 s	0.0058 s
GFLOPS		15.33	17.50

balance. As shown in Figure 7, a speedup of 1.14x is achieved for a 2^{16} -equation system, lower than for the case of cyclic reduction.

Table 1 shows a summary of the performance results we have obtained for the different implementations for a 2^{20} -equation tridiagonal system using single floating points arithmetic. The OpenMP cyclic reduction exhibits a speedup of 1.08x and Bondeli's algorithm 1.50x, with 0.44 and 0.78 GFLOPS, respectively. The GFLOPS rate for Bondeli's algorithm is nearly twice that for the cyclic reduction owing to the higher arithmetic requirements of the first as mentioned above.

The key for the OpenMP implementations is performing a good schedule that maximizes cache locality, especially in cyclic reduction, and selecting the optimal block size in Bondeli's algorithm to ensure a good workload balance. Memory cache plays an important role in the speedup of the cyclic reduction method. The best results for Bondeli's algorithm are achieved for large systems due to the memory requirements of the cyclic reduction that can not be satisfied by the multicore memory hierarchy.

5.2 CUDA

Figure 8 shows the speedups of the CUDA implementations for the cyclic reduction and Bondeli's algorithm varying the

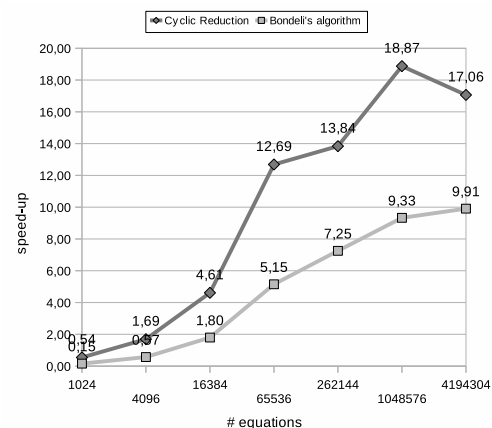


Fig. 8: CUDA speedup of cyclic reduction and Bondeli's algorithm with different tridiagonal system sizes.

system size. For small tridiagonal systems, the speedup is low, as the available parallel hardware cannot be fully exploited due to the small number of computations associated. For larger systems, the GPU shows good performance in terms of speedup. Both algorithms scale well as the system size increases. This is a typical behavior for the GPUs, which need to be fed with thousands of data to exploit their computational capabilities to the full [14]. A maximum speedup of 18.87x is achieved for the cyclic reduction implementation for a system of 2^{20} equations and a GFLOPS rate of 15.53 (see Table 1). A lower speedup is obtained by Bondeli's algorithm, with 9.3x for the same system size but a higher GFLOPS rate of 17.50. The reason is, as for the OpenMP implementations, the higher number of computations associated to Bondeli's algorithm.

6. Conclusions

We have analyzed two tridiagonal system solvers (cyclic reduction and Bondeli's algorithm) exhibiting different levels of parallelism in two different parallel hardware platforms: multi-core architectures (CPU) and GPUs platforms. The implementations are analyzed in terms of speedups and GFLOPS with respect to a sequential implementation of the Thomas algorithm. These algorithms were executed in a PC with an Intel Core 2 Quad Q9450 and a NVIDIA GeForce GTX 295.

In OpenMP, for small systems the cyclic reduction algorithm presents a better speedup than Bondeli's algorithm. This is mainly due to the high computational requirements of Bondeli's algorithm. When the system size increases over 2^{16} equations (large systems) the performance of the cyclic reduction implementation decreases as the data storage requirements for the algorithm exceed the capacity of the L2 cache, increasing the data movement. The speedup changes more linearly for Bondeli's algorithm because of the good workload balance and because it uses the Thomas algorithm which present sequential accesses to memory, for solving the subsystems generated by the algorithm. For systems of 2^{20} equations the best results are a speedup of 1.50x obtained by Bondeli's algorithm and a GFLOPS rate of 0.78.

The results in CUDA are always better for cyclic reduction. The complex algorithm structure and the different types of stages involved in Bondeli's algorithm makes the implementation thereof a challenge. For a 2^{20} equation system, the best results of the CUDA implementation are a speedup of 18.87x with a GFLOPS rate of 17.50.

The results have shown how the cyclic reduction solver, that exhibits fine-grained parallelism, achieves always best results in the GPU comparing to the Bondeli's algorithm, with a coarse-grained parallelism. This also happens in a multicore CPU when the system of equations was small enough to fit into the cache. When the size of the system is larger than 2^{20} , Bondeli's algorithm presents better results.

In conclusion, fine-grained parallelism problems often achieve highly positive results in the GPU, although the time

invested in adapting other types of problems also has its reward in terms of speedup.

ACKNOWLEDGEMENTS

This work was supported in part by the Ministry of Science and Innovation, Government of Spain, and FEDER funds under contract TIN 2010-17541, and by the Xunta de Galicia under contracts 08TIC001206PR and 2010/28.

References

- [1] L. H. Thomas, "Elliptic problems in linear difference equations over a network," *Watson Sci. Comput. Lab. Rept.*, 1949.
- [2] F. Gauss, "Theory of motion of heavenly bodies," 1809.
- [3] B. L. Buzbee, G. H. Golub, and C. W. Nielson, "On direct methods for solving poisson's equations," *SIAM Journal Numerical Analysis*, vol. 7(4), pp. 627–656, 1970.
- [4] R. W. Hockney, "A fast direct solution of poisson's equation using fourier analysis," *ACM*, vol. 12, pp. 95–113, 1965.
- [5] H. S. Stone, "An efficient parallel algorithm for the solution of a tridiagonal linear system of equations," *Journal of the ACM*, vol. 20, pp. 27–30, 1973.
- [6] S. M. . S. D. Müller, "A method to parallelize tridiagonal solvers parallel computing," vol. 17, pp. 181–188, 1991.
- [7] H. H. Wang, "A parallel method for tridiagonal equations," *ACM Trans. Math. Softw.*, vol. 7, no. 2, pp. 170–183, 1981.
- [8] S. Bondeli, "Divide and conquer: a parallel algorithm for the solution of a tridiagonal linear system of equations," *Parallel Comput.*, vol. 17, no. 4-5, pp. 419–434, 1991.
- [9] OpenMP Architecture Review Board, "OpenMP application program interface, Specification," 2008. [Online]. Available: <http://www.openmp.org/mp-documents/spec30.pdf>
- [10] NVIDIA, "Cuda technology," Nvidia Corporation, 2007. [Online]. Available: <http://www.nvidia.com/CUDA>
- [11] M. Harris, "General-purpose computation on graphics hardware," 2002. [Online]. Available: <http://www.gpgpu.org>
- [12] S. Allmann, T. Rauber, and G. Runger, "Cyclic reduction on distributed shared memory machines," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference*, vol. 0, p. 290, 2001.
- [13] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.
- [14] D. B. Kirk and W. m. W. Hwu, *Programming Massively Parallel Processors: a Hands-on Approach*. Massachusetts: Elsevier, Burlington, 2010.
- [15] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the gpu," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '10. New York, NY, USA: ACM, 2010, pp. 127–136.
- [16] D. Goddeke and R. Strzodka, "Cyclic reduction tridiagonal solvers on gpus applied to mixed-precision multigrid," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 22–32, 2011.
- [17] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 2:1–2:12.
- [18] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-d fft kernel for gpus using cuda," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 5:1–5:11.
- [19] "Tridiagonal system solvers, internal report," Department of Electronics and Computer Science, University of Santiago de Compostela, Santiago de Compostela, Spain," Technical report, 2011.
- [20] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide*, May 2011. [Online]. Available: <http://www.intel.com/products/processor/manuals>

Parallel Merge Sort Implementation Using OpenMP

Jaeyoung Park, Kyong-Gun Lee, and Jong Tae Kim
School of information communication engineering,
Sungkyunkwan University, Suwon, GyeongGi-Do,
South Korea

Abstract – One of representative sorting a algorithm, merge sort, is widely used in database system that requires sorting due to its stability. To raise performance of merge sort, it is effective to change parallel method but due to difficulty of changing database system, it requires parallel method that minimizes changing the system. We suggest parallel merge sort that uses OpenMP, capable of parallel, inserted directive to existing program code that improves performance while minimizing change to the system. Also, we examined performance of merge sort depending on k-th number of way and performance of parallel region occupied by sort that can affect sorting process by constructing variety of cases using 2^{20} or 1,048,576 data for experimental purposes. As a result, the most important factor that affects parallel method is usage of core, not ways. By considering this factor, experimental results demonstrate a speedup of 1.8 on 2 processor core, and a speed up of 2.8 on 4 processor core.

Keywords: OpenMP, Merge sort, k-way merge sort, parallel processing.

1 Introduction

For database system that processes large amount of information, there are many search algorithms that are fast and accurate for usage. Similarity of all these algorithms is that data are assembled in sort by search requirement [1]. It means that fast and accurate search is fast and accurate sorting. When huge amount of information is updated in a day, this situation demands more frequent data sort. Also, there is trend in which amount of data for sort is getting increased for one process. Therefore, in database system, time and effort are getting increased for sorting. A method is needed to reduce sorting time to fix this problem. A database system that processes huge amount of information needs increase in system effectiveness but since information must be provided continuously, it requires a method that does not alter system greatly while increasing effectiveness. However, a regular comparison-based algorithm can't go beyond $n \log n$ effectiveness change [2]. There are parallel method that takes care of data simultaneously in hopes of raising effectiveness but those methods arranges road balancing or makes communication between inter-processor faster[3], which makes construction of parallel method difficult and creates

great change to the system in the process. Trying to solve these weaknesses, this thesis chose merge sort algorithm due to its characteristic of being stable, which is a reason it is widely used in database system, and OpenMP was used for parallel method. OpenMP would not alter system greatly by inserting directive for parallel code into existing code, and it is supported by most of compilers so it can be used in most of systems [5]. Therefore, parallel merge sort using OpenMP can solve problems discussed earlier. In comparison to other methods, it can be implemented easily so we can expect higher effectiveness without much effort. Along with parallel merge sort algorithm, variety of scenarios will be constructed so that results will be analyzed by examining effectiveness depending on change of k in k-way merge sort and effective of parallel region in applying OpenMP to suggest effective parallel method.

Section 2 would explain k-way merge sort algorithm and OpenMP that was used for parallel sorting. Section 3 would explain which method to divide data domain in attempt to parallel sorting. Next, Section 4 would analyze the results and finally give conclusion.

2 Related Work

2.1 K-way Merge Sort

Merge sort is one of premium sorting algorithms that before sorting, data is separated (divide), each partial component is sorted (conquer), then all sorted components are pasted together in a cycle to sort entire data, which is one of the (divide and conquer) method. During a cyclical call, it goes through three steps; divide, conquer, and paste. Divide is a step where given data is divided into many small data. Conquer is a step where divided data is sorted in orderly manner. Last step is paste where data is pasted and completely sorted. During merge sort's process, depending on how unit is divided and goes through those three steps are called k-way merge sort. Like Fig 1, if it is divided into two units, it is called 2-way. Like Fig 2, if it is divided into four units, it is called 4-way.

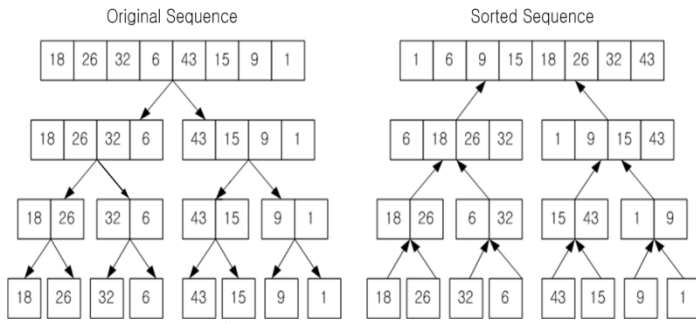


Figure 1. 2-way merge sort

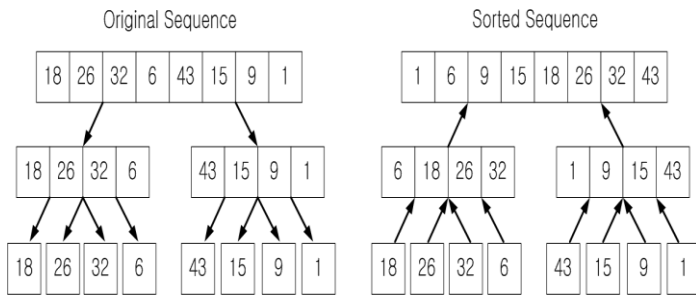


Figure 2. 4-way merge sort

2.2 OpenMP

OpenMP is an API[6] for parallel programming model that allowed to be used by application by inserting compiler directive to allow parallel processing. Even if program code is created in order manner, if directives are added, only specified area is changed for parallel processing. OpenMP can only be used is SMP (shared memory multiprocessor) but after few years, most of CPU provides at least dual core so there is no problem for use it [7]. Its positive trait is that it is de-facto standard, therefore it is provided by many compilers. It can be inserted into given program code so with little effort we can expect high rise in effectiveness and expansion. Fig 3 is the example of the use of OpenMP. When executing the below instruction, the execution time can be reduced by parallel add operation with 4 threads.

```
#pragma omp parallel for shared(n,a,b,c) private(i)
for(i=0;i<n;i++)
c[i]=a[i]+b[i];
```

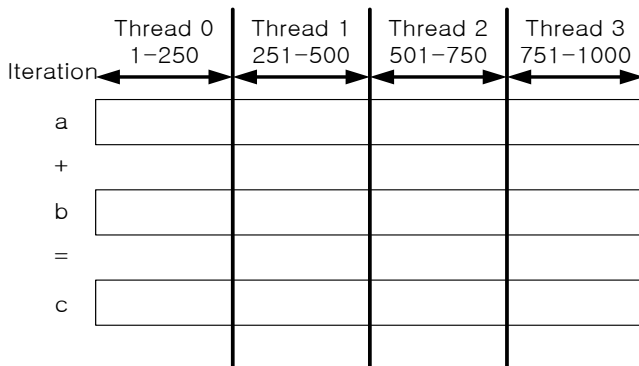


Figure 3. Example of the use of OpenMP

3 Parallel Method

In this paper, previously mentioned merge sort is paralleled by using OpenMP. In types of merge sort, to find out difference in speed due to number of way, way is changed and implemented to 2-way, 4-way, and 8-way. Also, to find out difference between performance of parallel region and performance due to number of cores, according to number of way, method of dividing and number of cores were changed together.

3.1 2-way Merge Sort

In case of Fig 4, total data is divided into four (1). Divided data are each goes to merge sort in (2). In (3) and (4), it goes through merge sort again and then pasted into sorted data. In a case where four cores are used, in (2), each one goes through merge sort so (2) uses four cores, and (3) and (4) each uses two and one respectively. When two cores are used for parallel processing, in (2), first two data are sorted then next remaining two are sorted. Then it goes to (3). Number of core used in (2) is two, and it was used twice. (3) and (4) used two and one respectively to reduce the time.

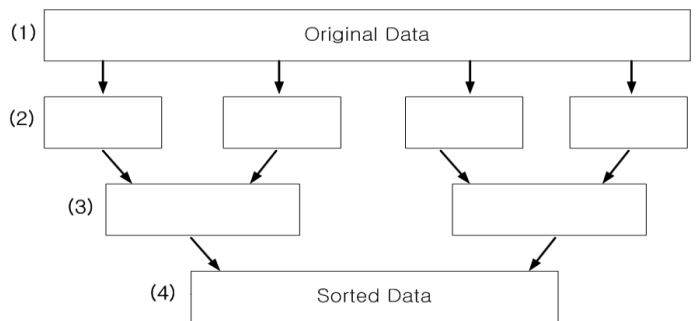


Figure 4. 2-way merge sort divided by 4 data

In case of Fig 5, it is total process is equal to Fig 4, but the data is divided into 8 units and if four cores are used, in (2), first four goes through merge sorted, then next four goes through merge sort. In (3), eight data are simultaneously pasted together using four cores. In case where two cores are used, in (2), two data goes through merge sort four times. In (3) and (4), two cores are used twice and once respectively. In last step (5), only one core is used. This was for comparing number of division and speed of sorting when total amount of data is equal for merge sort.

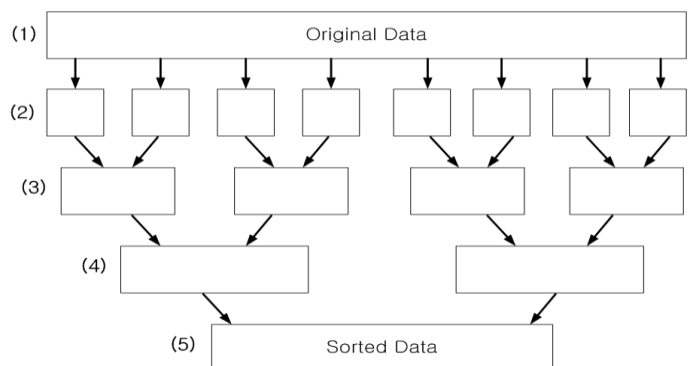


Figure 5. 2-way merge sort divided by 8 data

3.2 4-way Merge Sort

4-way merge sort, different from 2-way which compares two data at a time, it can compare four data at once for sorting. Therefore, data distribution is increased to multiples of four, four and sixteen, to compare four data at once. Fig 6 shows entire data is divided into four so one through four data goes through merge sort by each core, and it goes through merge sort in the last step. If all four cores are used, it can go to last step at once. If two cores are used, parallel processing goes twice in the second step and one core is used for the last step in the 4-way merge sort. Fig 6 shows order for 4-way merge sort when data is divided into four. In case where two cores are used, dark box shows data pairing where 1 and 2 are processed, then 3 and 4, then when 1 through 4 is done, last 4-way merge sort is done by using only one core.

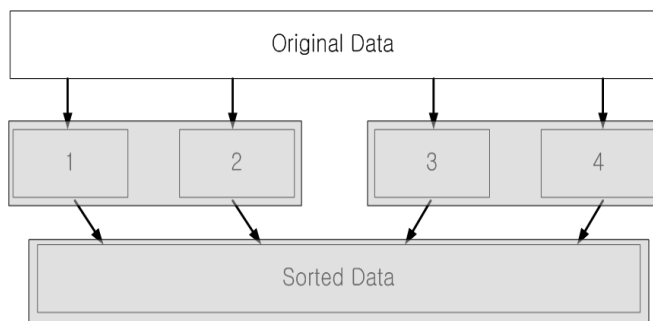


Figure 6. 4-way merge sort divided by 4 data

Fig 7 shows 4-way merge sort when data is divided into sixteen. Merge sort is processed in numerical order, and dark box shows data pairing when four cores are used. Four cores each process 1~4, 5~8, 9~12, and 13~16 then process 17~20. After that you get the result in the last step with one core. If only two cores are used, then units should be numerically paired in two and be processed.

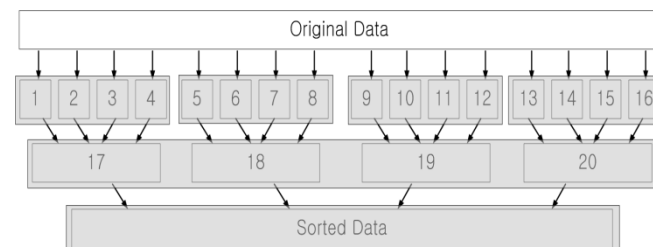


Figure 7. 4-way merge sort divided by 16 data

3.3 8-way Merge Sort

If you ignore the fact that number of units that can be processed at once is eight, then 8-way merge sort is similar to other k-way merge sort. Fig 8 shows method for arranging data by dividing it into eight. Once 1~8 are sorted by merge sort, then 8-way merge sort is processed once more to get final result.

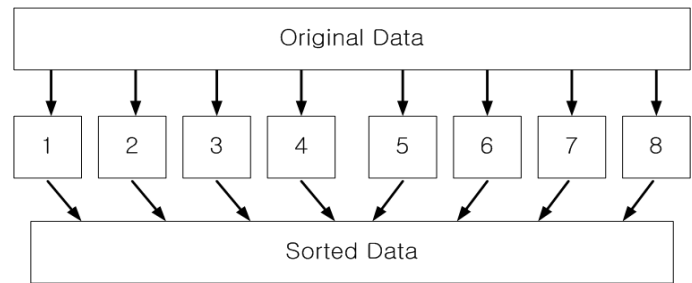


Figure 8. 8-way merge sort divided by 8 data

4 Experimental Results

Experiment took place in quad core 2.4Ghz environment. 2^{20} or 1,048,576 data were randomly generated and measured time it took to sort. Time is measured using CPU internal clock. Table 1 results are averages of 100 times it ran to minimize error that can occur using CPU internal clock. Unit for time is sec. Horizontal line shows number of cores processing merge sort and vertical line shows under what condition merge sort is processed. It also shows number of data divided for 2-way, 4-way, and 8-way. One means one data is used, and it was not divided. For each k-way, one means no parallel process exist, therefore no speed up.

Table 1 results shows that in 4-way and 8-way, even if there were no parallel process, in case of 4-way when data is divided into four and in case of 8-way when it is divided into eight, time was reduced slightly. Through running it 100 times, it showed steady difference so it can't be said that it is an error. It can be predicted that it happened because experiment was done in quad core system so it can be effect of optimizing compiling stage and effect of multi thread that is automatically done by operating system. It's increases is too small, and it is in margin of error in 2-way so it clearly does not show improvement so we can't conclude that dividing earlier data into many pieces leads to improvement.

In looking at method of dividing data for parallel processing in k-way, if two cores are used, when data is split into four you get 0.1749. When data is split into eight you get 0.1753. If four cores are used, when data is split into four you get 0.1162. When data is split into eight you get 0.1192 so it is almost equal. In 4-way, you get 0.2042 and 0.1971 for two cores and 0.1344 and 0.1351 for four cores. Therefore raising number of dividing data does not much affect parallel performance. This is due to four or two core are run with limits of CPU so CPU usage is almost same. If in 2-way, where data is divided by two, CPU that actually runs it can't use more than two cores so as shown in Table 1, you can't expect improvement when four cores are used. Also if in 4-way, when data is split into five, previously sorted four must wait for the fifth one so it would reduce effectiveness. However, in 2-way where data is divided into two, although one would expect no improvement, it improved effectiveness by 1.17. This is a small profit of work being processed earlier by optimal condition in compiling stage. Therefore, there is no change in conclusion that if there is N core, then data should be divided in multiples of N for most improvement in effectiveness.

Table 1. Execution time of merge sort (sec)

Num of cores		Single	Dual	Quad
Condition				
2-way	1	0.3362	0.3376	0.3377
	2	0.3405	0.2076	0.1768
	4	0.3345	0.1749	0.1162
	8	0.3396	0.1753	0.1192
4-way	1	0.3857	0.3849	0.3832
	4	0.3788	0.2042	0.1344
	16	0.3614	0.1971	0.1351
8-way	1	0.3936	0.3915	0.3906
	8	0.3669	0.2035	0.1246

Most clearly shown result is when core is increased to take care of many data simultaneously. If two cores are used for parallel processing, in case of 2-way with dividing data into four showed above 1.9 improvement. If four cores were used it showed 2.8x improvement. It used twice as much cores but showed 1.5x improvement from two. This is due to parallel region is limited which all four cores can't participate in processing at once. In 8-way, from single to dual showed 1.8x improvement and single to quad showed 2.9x improvement similar to 2-way. In case of 4-way, if data is split into four and sixteen, single to quad showed 2.8x and 2.67x respectively. This is due to process where data is increased for single core, some of it were sorted early by operating system. So area available for OpenMP were smaller. Specific number is different but shows similar amount of improvement in performance.

5 Conclusion

For improving performance of database system, performance of sorting algorithms must increase. Among techniques that improve performance, there is parallel processing. However, it is difficult to materialize, unexpected error can occur, and can alter system greatly. Therefore, being effective and easy to use, we paralleled merge sort algorithm using OpenMP. We got the results like table 1 changing the number of cores which participate merge sort process and the region of parallel processing. When looking at the results using the same number of cores, there is no clear relationship with k-way and merge sort. Though changing k-way, it can reduce number of combining but due to computer's trait, k can't be all handled at once and internally it uses binary comparator so total number of comparison is the same. It should be distributed to parallel merge sort algorithm using OpenMP and parallel region should be at least be greater than

number of core so that it can reduce time that core is not being used. If the number of parallel regions are less than the number of cores, the performance is hardly improved not being able to conjugate the resources. It can be shown by using single core with k-way merge sort and using dual or quad core with 2-way merge sort. For greater effectiveness, area should be in multiples of number of core so that when earlier steps are done, number of core waiting can be reduced, which will result in greater usage of entire core and great performance in parallel processing. Through running merge sort algorithm we implemented with conclusions, in case of dual core, it showed 1.8x improvement, and in case of quad core, it showed 2.8x improvement. Finally, without changing system greatly, data should be divided so that area where OpenMP is applied should be set and parallel that part so great improvement were acquired.

6 References

- [1] Gary G, Leonard D, Sujata R. "Parallel Merge Sort Method and Apparatus". United States Patent.
- [2] VLADIMIR E, DERICK W. "A Survey of Adaptive Sorting Algorithm"; ACM Computing Surveys., Vol. 24., No. 4., December, 1992.
- [3] Minsoo K, and Dongseung K. "Parallel Merge Sort with Load Balancing"; International Journal of Parallel Programming., Vol. 31., No. 1., 2003.
- [4] Knuth D.E. "The Art of Computer Programming"., Vol. 3:Sorting and Serching., Addison-Wesley., 1973.
- [5] Rohit C, Leonaldo D, Dave K, Dror M, Jeff M, Ramesh M. "Parallel Programming in OpenMP", 2001.
- [6] Bakara C, Gabriele J, and Ruud van der P. "Using OpenMP", 2007.
- [7] Leonardo D, Ramesh M. "Standard API for Shared-Memory Programming"; IEEE Computational Science & Engineering, 1998.

Low-synchronisation Work Stealing under Parallel Data-List Processing in Multicores

J. Buenabad-Chávez¹, M. A. Castro-García², J. L. Quiroz-Fabián², D. M. Yellin³,
G. Román-Alonso² and E. F. Hernández-Ventura¹

¹Departamento de Computación, CINVESTAV-IPN, México City, D.F., 07360 México

²Departamento de Ing. Eléctrica, UAM-I, México City, D.F., 09340 México

³IBM Israel Software Lab, Jerusalem Tech Park, Jerusalem, 96951 Israel

Abstract—*In the context of processing data lists in parallel in a multicore, various threads share a workload, each using a list to get and insert the data items to be processed; and when a list becomes empty, the owner thread steals data items from another list — thus balancing the workload according to the processing capacity of each thread transparently to the programmer. The first algorithm we designed to synchronise work stealing in such context addressed various important issues such as cache locality and memory consumption.*

This paper presents a new low-synchronisation algorithm for work stealing in the above context which improves performance and also tends to make performance more predictable. We present experimental results comparing both algorithms using various applications with different granularities and access patterns.

Keywords: Parallel Computing, Load Balancing, Multithreading, Shared Memory, Atomic Operations

1. Introduction

The arrival of multicores made parallel computing mainstream, meaning that performance in these architectures is to be mostly improved both through multithreaded parallelism based on shared memory communication and through mechanisms to load balance the workload among the threads running an application. This is not an easy task, and thus software tools and middleware have been proposed whose aim is to hide parallelism and load balancing issues from end users. OpenMP can be used to make sequential code parallel through the use of keywords and without affecting the sequential source code. However, this approach is mostly of benefit for sequential end-user applications.

Middleware for parallel computing, such as Skeletons [1], [2] and Mapreduce [3], must be modified to capitalise on the inherent parallelism of multicores typically within the context of clusters of multicore nodes, addressing both intra- and inter-node parallelism through shared memory and message passing, respectively. This is the context of our work. We outline this context below. The work presented here is, however, only (or mostly) relevant to multicore platforms, and has various points in common with the “data

structures in the multicore age” presented in [4]; we discuss some of those points in Section 6.

We have developed the Data List Management Library (DLML), a middleware to process data lists in parallel. Users only need to organise their data into items to *insert* and *get* from a list using DLML functions. DLML applications run under the SPMD (Single Program Multiple Data) 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 processing in all nodes is over. 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 [5], [6] was targeted at clusters composed of uniprocessor nodes, and was 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 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.

Although the first version of DLML can run unmodified in clusters of multicore nodes (through running an application process and a DLML process for each core), we obviously wanted to capitalise better on the inherent parallelism of each multicore by reducing the communication overhead of message passing between sibling cores through multithreaded parallelism and communication based on shared memory. This proved to be more difficult than we initially thought, however. The first version of DLML would run much better than our initial multithreaded designs. After addressing cache locality and memory consumption issues, we came up with a multithreaded design that performed better than the first version of DLML. This multithreaded design uses *global locking* for list refilling: when an empty

list is to be refilled through stealing data items from another list, all lists in a multicore are locked.

This paper further improves on our multithreaded design and presents a *low-synchronization locking* algorithm for work stealing in order to refill an empty list. Basically, this algorithm does not require all threads to acknowledge synchronisation through identifying safe phases to access lists. This algorithm not only improves performance but also tends to make it predictable. We compare our low-synchronization algorithm to our global-locking algorithm using various applications with different granularities and access patterns. We do this only in the context of a single multicore (not clusters) to appreciate better the behaviour of both algorithms (as in clusters the overhead of message passing becomes dominant).

Section 2 presents some of the design issues addressed in the design of DLML for multicore nodes. Section 3 describes our *global locking* algorithm and our *low-synchronization locking* algorithm. Section 4 presents our experimental platform and applications to compare both algorithms. Section 5 presents our results. Section 6 presents related work and we conclude in Section 7.

2. Designing a Multicore DLML

This section describes our experience in designing a multicore version of DLML based on multithreaded parallelism and shared memory for intra-node communication. In so doing, we will refer to the original DLML based on multiprocess parallelism and message passing as DLML, and to the *MultiCore* (MC) version(s) as MC-DLML.

We designed various versions of MC-DLML before the version that outperformed DLML. The first versions of MC-DLML were outperformed by DLML by a significant margin, particularly running our fine-grain application, despite the fact that DLML uses messages to send list items between sibling (in-same-node) DLML and application processes. We eventually realised we had to address four issues: i) the locking overhead for MPI calls to be thread safe, ii) cache locality, iii) memory consumption, and iv) intra-node synchronisation cost.

i) Although the first issue is not core to the main algorithm described in this paper, we briefly discuss it for completeness. One early design of MC-DLML used one thread to make (data) requests to remote nodes and another thread to serve requests from remote nodes, and as many application threads as the number of cores available in each multicore node. The coding was simpler to understand and thus to maintain. Another design used that many application threads and another thread both to make and to serve requests; this thread would make remote requests through messages after receiving, through messages too, *local requests* from sibling application threads. However, allowing two or more threads to make MPI calls 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. Thus our design of MC-DLML that outperforms DLML uses only application threads and only one of them, the one that initialises MPI, makes MPI calls under `MPI_THREAD_FUNNELED` support to make/serve requests to/from remote nodes.

ii) We also tried various alternatives regarding list management within each multicore: 1, 2 and even 3 lists for each application thread to reduce contention. One version used a list to get items and another dynamic list to insert items dynamically generated; items from dynamic lists were moved to a global list from which other threads could steal work when their lists became empty. However, performance was bad with fine granularity because we were losing cache data locality, as follows. DLML manages a single list for each application process: getting items from, and inserting items dynamically generated at, the front of the list. Hence items that have just been placed on the list are sooner removed from the list and processed while still resident in the cache. By using a separate list for dynamically generated items we lost this cache locality.

iii) Also, one of our applications, the non-attacking Queens (NAQ) problem, consumes much less memory when we use a single list for getting and inserting elements as described above. The NAQ problem consists of finding all possible ways of placing N queens on an $N \times N$ chessboard, so that no queen attacks another queen [7]. The parallel version is achieved by placing one more queen at a time and inserting the new item (with one more queen) into the list so that other threads can insert another queen until either all queens have been placed or no more queens can be placed because they attack other queens. Thus an item with more queens has a shorter lifecycle – it is closer to N queens. Hence by placing more recent items at the front of the list – those with more queens – less memory will be consumed, and the algorithm will perform more efficiently and require less swapping if at all (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, and list refilling using a 2-phase global locking. The rest of the paper will focus on (iv) how to lower the synchronization cost of this global locking.

3. Work Stealing within each Multicore

3.1 Global Locking Work Stealing

Our design of MC-DLML that outperformed DLML uses the work stealing algorithm described in this section for list refilling within each multicore. The context is as follows: each application thread has its own list to get and insert items. The work-stealing algorithm operates entirely within the procedure `DLML_get()`.

```

1 int DLML_get( LIST *L, ITEM *item ) {
2   if stoppedthreads > 0
3     // wait if a refill is going on
4     increase_byone_atomic( stoppedthreads )
5     copy = refillstamp
6     wait while copy == refillstamp
7   if L->size > 0
8     *item = L->first      // GET ITEM
9     ...
10    return 1
11  else                          // REFILL LIST
12    increase_byone_atomic( emptylists )
13    __1st_LOCKING_PHASE__:
14    lock( refilllock )
15    stoppedthreads = 1    // 2nd locking phase
16    wait until:
17    stoppedthreads+emptylists == THREADS_NR+1
18    loop over other lists to choose largest
19    if data available
20      refill L taking half of largest
21      stoppedthreads = 0
22      refillstamp++
23      decrease_byone_atomic( emptylists )
24      unlock( refilllock )
25      return DLML_get( L, item )
26    else if emptylists < THREADS_NR
27      stoppedthreads = 0
28      refillstamp++
29      unlock( refilllock )
30      goto __1st_LOCKING_PHASE__ // try again
31    else
32      stoppedthreads = 0
33      refillstamp++
34      unlock(refilllock)
35      return 0                // finish
36 }

```

Fig. 1: Global-locking work stealing.

Figure 1 shows the main tasks of `DLML_get()` (in a multicore) in pseudo-code: getting an item and refilling an empty list. Getting an item does not require locking (lines 7-10 in that figure), nor does inserting an item. Refilling an empty list does require locking all lists, as it involves choosing the largest list and moving (stealing) half of its items to the list being refilled. To refill its list a thread must: i) acquire `refilllock` (line 14), ii) signal other threads to stop using their lists, through turning on the flag-and-counter `stoppedthreads` (line 15), and iii) wait for all other threads to stop using their lists, a condition that is reached when `emptylists+stoppedthreads == THREADS_NR+1` (lines 16-17), as follows. Other threads either will be trying to refill their lists, and thus have atomically increased `emptylists` by one and are blocked in `refilllock` (lines 12-14), or will be processing an item and will next call `DLML_get()` and find `stoppedthreads > 0` (line 2), and will then increase `stoppedthreads` by one and then wait while `copy == refillstamp` (lines 4-6). In the condition in lines 16-17, `THREADS_NR+1` is used as opposed to `THREADS_NR` because the thread refilling its list has both increased `emptylists` and turned on the

flag-counter `stoppedthreads`.

Once all other threads have stopped using their lists, the thread refilling its list loops over all other lists to choose the largest one (line 18). If it finds data available, it refills its list by moving half of the items of the largest list, undoes both locking phases, decreases `emptylists` by one, and finally calls `DLML_get()` again (lines 19-25); this call will succeed in returning an item (to the calling application code) because the list has just been refilled.

If the thread refilling its list finds **no** data available (i.e., finds all other lists empty after looping over all of them), it checks this condition again through checking the value of the variable `emptylists` (line 26). This variable is increased by one by each thread that has exhausted its list (line 12) and is trying to acquire `refilllock` (line 14) to start a refill operation. If `emptylists` is smaller than the number of application threads (`THREADS_NR` in line 26), there may be threads whose list is empty but may be processing an item from which new items will be generated (one of our applications generates new items dynamically); hence the thread refilling its list undoes both locking phases and goes to `__1st_LOCKING_PHASE__` in line 13 to start the refill operation again (lines 27-30). On the other hand, if `emptylists` is not smaller than `THREADS_NR` but equal (lines 31-35), then all data has been exhausted and all threads are trying to refill their lists; each thread reaching this condition undoes both locking phases and returns 0 to the application code, signalling the end of computation.

3.2 Low-synchronisation Work Stealing

The main problem with global locking for work stealing is that a thread refilling its list must wait for all other threads to acknowledge synchronisation (i.e., stop using their lists) before it can start refilling its list. This is an issue with coarse grain applications or if imbalance occurs out of other applications sharing the cores within a multicore, as in either case some of the threads running under DLML will not acknowledge that synchronisation immediately, thus delaying each list refilling. The algorithm presented in this section has low synchronisation overhead by not requiring all threads to acknowledge synchronisation; this is accomplished through identifying safe phases to access the lists of other threads.

Figure 2 shows `DLML_get()` with low-synchronisation locking in pseudo-code. Refilling a list is also based on 2-phase locking, and the overall structure of the algorithm is the same as that of the global locking algorithm in Figure 1; only the lines marked `/**` are different or new. The first locking phase is the same as with global locking: acquiring `refilllock` (lines 14-15). The second phase is somewhat different, however: after turning on the flag `stopthreads` (line 16) to signal that a refill is going on, the thread refilling its list does not wait for all other threads to stop using their lists as in the global locking algorithm. Instead, it immediately loops over all other lists to choose the largest

list L2 that is *not being used*, a condition known to hold as described below. This algorithm is quicker to refill than the global locking algorithm out of not waiting for all other threads to stop using their lists, but may not choose the largest list as the global locking algorithm does.

A list L2 is *not being used*, and will not be used, by the owner thread if any of the following two conditions holds: either `L2->inside_dlml == 0` or `L2->stamp==refillstamp` (lines 17-18), as follows. Each thread that enters `DLML_get()` increases by one its flag `inside_dlml` in line 2, thus signalling it is using its list; this signal is turned off just before the thread returns from `DLML_get()`: `inside_dlml` is decreased by one (to 0) or assigned the value 0 in lines 10, 21, 33. Likewise, each thread that enters and returns from `DLML_insert()` (not shown) increases and decreases by one its flag `inside_dlml`. Thus, if `L2->inside_dlml == 0` (lines 17-18), the owner thread of L2 is *not* running within `DLML_get()` nor within `DLML_insert()` (but is executing application code), and it is safe to use its list for refilling.

If `L2->inside_dlml` is not equal to 0, the owner thread of L2 is running within `DLML_get()` or `DLML_insert()`. However, it may be waiting for the refill going on to finish after having acknowledged the `stopthreads` signal (lines 3-6). If this is the case, `L2->stamp==refillstamp` (lines 17-18), and it is also safe to use its list for refilling.

`DLML_insert()` is not shown but its first 5 lines of code are the same as the first 5 lines of code of `DLML_get()` (lines 2-6). Also, the last line of code of `DLML_insert()` is the same as the 10th line of code of `DLML_get()`, which atomically decreases by one `L->inside_dlml` (to 0), thus signaling the running thread has returned to application code.

We still use the variable `emptylists` (lines 13, 24, 27), in the same way as in the global locking algorithm, to detect when all lists are empty in order to signal the end of computation. Note that the use of `increase_byone_atomic()` operations with variables `L->inside_dlml` in line 2 and `stopthreads` in line 16 is compulsory in architectures that are not sequential consistent in order to commit immediately the new value to memory, thus serialising the access to both lists and avoid race conditions. If a thread refilling its list used `stopthreads = 1` instead, the new value would be stored only in the cache; another thread would not see it and would proceed to access its list L, while the former thread would have proceeded to loop over all other lists and could eventually take elements from L. This simultaneous manipulation of L would not be safe. In lines 21-22, 28 and 33, there is no need to use atomic operations on those variables because the new values are committed to memory on unlocking `refilllock` (a few lines below).

```

1 int DLML_get( LIST *L, ITEM *item ) {
2   increase_byone_atomic(L->inside_dlml) // ***
3   if stopthreads > 0
4     // wait if a refill is going on
5     L->stamp = refillstamp
6     wait while L->stamp == refillstamp
7   if L->size > 0
8     *item = L->first           // GET ITEM
9     ...
10  decrease_byone_atomic(L->inside_dlml) // ***
11  return 1
12  else // REFILL LIST
13    increase_byone_atomic( emptylists )
14    __1st_LOCKING_PHASE__:
15    lock( refilllock )
16    increase_byone_atomic( stopthreads ) // ***
17    loop other lists & choose largest list L2 if:
18    L2->inside_dlml==0 || L2->stamp==refillstamp
19    if data available
20      refill L from half of L2
21      L->inside_dlml = 0           // ***
22      stopthreads = 0
23      refillstamp++
24      decrease_byone_atomic( emptylists )
25      unlock( refilllock )
26      return DLML_get( L, item )
27  else if emptylists < THREADS_NR
28    stopthreads = 0
29    refillstamp++
30    unlock( refilllock )
31    goto __1st_LOCKING_PHASE__ // try again
32  else
33    L->inside_dlml = 0
34    unlock( refilllock )
35    return 0 // finish
36 }

```

Fig. 2: Low-synchronisation work stealing.

Line 10 could be `L->inside_dlml = FALSE` instead of `decrease_byone_atomic()`, but the latter would make the relevant list immediately visible for refilling another list.

4. Experimental Platform and Applications

This section describes the experimental platform and applications we used to compare the performance of both algorithms described above. Our hardware platform is a cluster with 32 multicore nodes. However, recall that we will only show experiments within a single multicore node to appreciate better the performance of both algorithms, as using two or more nodes would involve message-passing overhead which will become dominant. Each node has a 2.67GHz Intel i7 920 processor with 4 hyperthreading cores, 4GB RAM, and a 500 GB disk. The software library used was the Pthreads library v. 4.1.2.

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 [8], [9], 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 $N = 300$ and 400 .

The NAQ problem consists of finding all possible ways of placing N queens on an $N \times N$ chessboard, so that no queen attacks another queen [7]. The search space of NAQ can be modeled 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.

5. Results

This section presents experimental results on the performance of global locking work stealing (global-locking) and low-synchronisation work stealing (lowSync-locking) 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. Note that in our hardware platform each node has 4 hyperthreading cores which appear to software (calling `sysconf(_SC_NPROCESSORS_ONLN)`) as being 8 cores.

5.1 Image Segmentation (IS)

Figure 3 shows the performance of global-locking lowSync-locking running IS with 165 image cuts. IS is a static application in that the total number of images is inserted into a list at the start of computation by the master thread, and is coarse-grain in that processing each image cut is relatively expensive compared to getting its id from a list. IS gets itself imbalanced due to the different costs of processing image cuts with different pixel intensities; hence work stealing does occur, and lowSync-locking performs better than global-locking the more number of threads are used. The fewer the number of threads used, the more work each thread has to do and thus the less work stealing occurs. As the number of threads increases, each thread has less

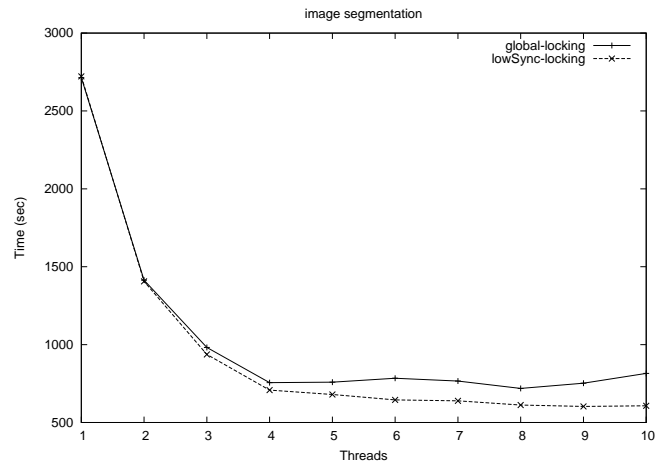


Fig. 3: Global-locking and lowSync-locking work stealing for IS.

work to do and more work stealing occurs. With global-locking, this means that each work stealing action has to wait for all other threads to finish processing the current image cut they are processing in order for them to be able to acknowledge the synchronisation required by global locking. With 8 threads the performance gain of lowSync-locking is about 100 seconds. Note that using more than 8 threads does not affect performance with lowSync-locking but does with global-locking. We analyse this further below.

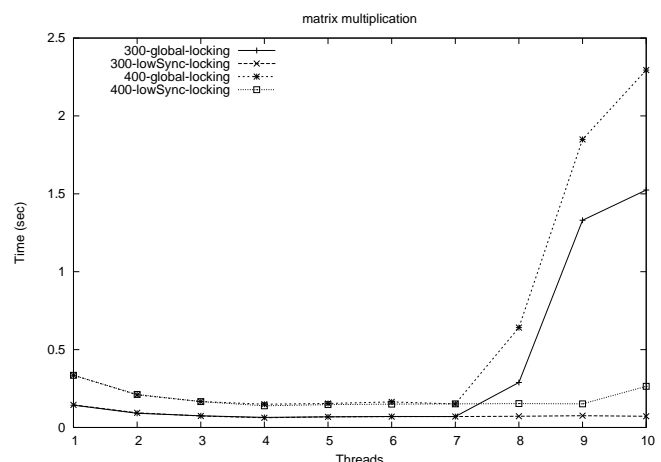


Fig. 4: Global-locking and lowSync-locking work stealing for MM.

5.2 Matrix Multiplication (MM)

Figure 4 shows the performance of global-locking and lowSync-locking running MM with $N = 300$ and 400 . MM is a static, medium-grain application. Global-locking and lowSync-locking perform similarly using from 1 to

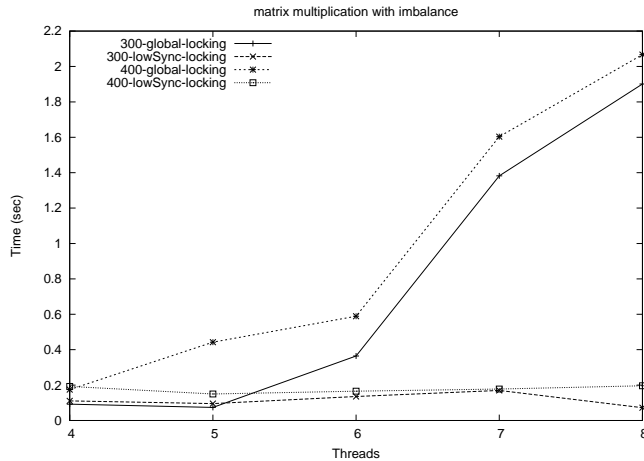


Fig. 5: Global-locking and lowSync-locking work stealing for MM with *imbalance*.

7 threads. With more threads the performance of global-locking deteriorates significantly. MM runs for a very short time (less than 1s with no imbalance) and thus any of its threads that gets blocked significantly affects overall performance. This is for the following reason: if while a thread is suspended another thread attempts a refill, global-synchronization will cause all threads to wait and synchronize, and this synchronization cannot happen until the suspended thread becomes live again. This is the case when more than 7 threads are used as at least one of them will be temporarily suspended by the operating system threads that run periodically.

This effect can be seen more easily when we create an external load on the machine, which will cause some of the threads of MM to block. Figure 5 shows runs of MM with an external load consisting of 4 threads continuously running within a while cycle computing sine and cosine values. This external load (4 threads) is applied to MM running with 4 to 8 threads, and thus contention for the cores available increases with the number of threads run by MM. The performance of global-locking deteriorates accordingly, as each work stealing action becomes more costly out of having to wait for suspended threads to acknowledge the synchronisation required by global locking. Note that global-locking performs slightly better than lowSync-locking for $N = 300$ with 4 and 5 threads, wherein contention for cores is not too high. The reason for this is the extra overhead of using atomic operations by lowSync-locking to signal a thread getting in and out of DLML.

5.3 The Non-attacking Queens (NAQ) problem

Figure 6 shows the performance of global-locking and lowSync-locking running NAQ for 14 – 16 queens (with no imbalance). NAQ is a fine-grain application: processing an item (usually adding a queen) is very quick compared

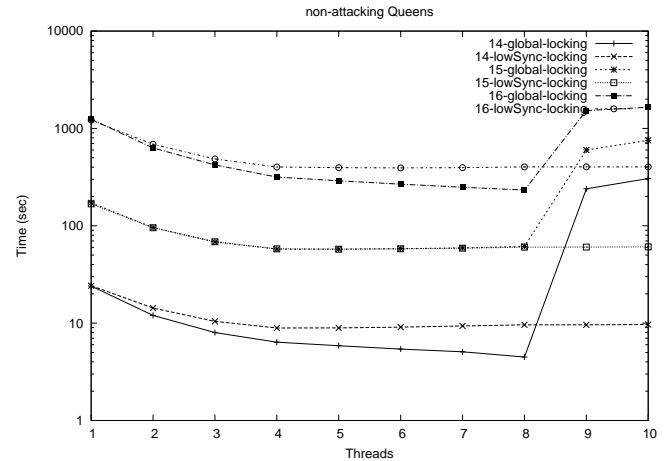


Fig. 6: Global-locking and lowSync-locking work stealing for NAQ – time in logarithmic scale.

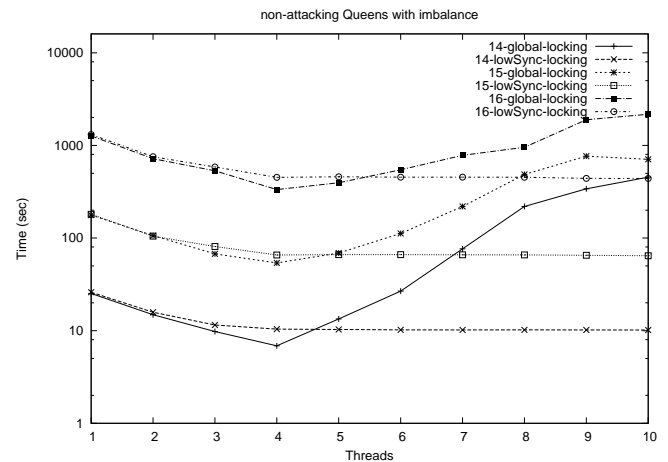


Fig. 7: Global-locking and lowSync-locking work stealing for NAQ with imbalance – time in logarithmic scale.

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. It also becomes imbalanced due to dynamically generating data. The figure shows that for long-running computations (16 queens) the use of atomic operations by lowSync-locking to signal getting in and out of DLML can incur a high overhead (the time scale is logarithmic). However, lowSync-locking is more stable than global-locking when there is contention for the cores available, as shown when more than 8 NAQ threads are used, or when there is external imbalance (consisting of 4 more threads as described above), as shown in Figure 7 (the time scale is also logarithmic).

6. Related Work

Work stealing is a scheduling algorithm and is discussed in [10] in the context of scheduling multithreaded computations in multiprocessors, and in context of hierarchical systems (clusters and grids) in [11], which compares two recent systems that use work stealing: Satin and Kappi. With *work stealing*, whenever a processor is underutilised, it attempts to steal threads/tasks from other processors. In contrast, with *work sharing*, “whenever a processor generates new threads, the scheduler attempts to migrate some of them to other processors in hopes of distributing the work to underutilized processors... Intuitively, the migration of threads occurs less frequently with work stealing than with work sharing, since when all processors have work to do, no threads are migrated by a work-stealing scheduler, but threads are always migrated by a work-sharing scheduler” [10]. Our work is based upon work stealing of data items. Hence it is perhaps best to call this work *data stealing*. In DLML, all application threads within a multicore run the same code, and we have not addressed any issues relating to thread scheduling yet, but rather reducing communication between threads under work stealing actions.

Our work so far requires that the processing of each data item be independent from the processing of any other data item. As such, our work is particularly relevant to the design issues of data structures for multicores discussed in [4], wherein techniques are suggested to reduce communication overhead between threads in multicores. Various techniques are suggested including low-contention locking, an elimination tree, ... and a *relaxed* distributed data structure with no ordering of operations for the sake of performance. This relaxed distributed data structure is very similar to the various data lists we managed (one for each thread) for the purpose of reducing contention and improving cache data locality. In the design of MC-DLML we tried several variations of those techniques, e.g., a single list with low-contention locking. However, as mentioned earlier, contention was high with fine-grain applications and we also were losing cache locality. Overall our work provides empirical evidence to some of the design issues discussed in [4].

7. Conclusions

We have presented a low-synchronisation algorithm for work stealing in the context of parallel data list processing in multicores. Our algorithm shows good performance overall and is more stable than the first algorithm we designed based on global locking. Low synchronisation is achieved by not requiring all threads to acknowledge synchronisation when a work stealing action is going on; this in turn is achieved through identifying safe phases to access other threads lists. Identifying safe phases is based on the use of atomic operations whose use under long-running fine-grain applications incurs a significant overhead. We will address

this issue in the future; we will also investigate concurrent list refills as opposed to only one list refill at a time as is the case with our current implementation. Our work has various points in common with the design issues of data structures for multicores discussed in [4], and we will investigate some of them further in the context of DLML for multicores.

References

- [1] 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.
- [2] H. Tanno and H. Iwasaki, “Parallel skeletons for variable-length lists in sketo skeleton library,” in *Euro-Par*, 2009, pp. 666–677.
- [3] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” in *Operating Systems Design and Implementation*, 2004, pp. 137–149.
- [4] N. Shavit, “Data structures in the multicore age,” *Commun. ACM*, vol. 54, no. 3, pp. 76–84, 2011.
- [5] 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.
- [6] M. A. Castro-García, “Programación con listas de datos para cómputo paralelo en clusters,” Ph.D. dissertation, CINVESTAV-IPN, México 07360, D.F., 2007.
- [7] A. Bruen and R. Dixon, “The n -queens problem,” *Discrete Mathematics*, vol. 12, pp. 393–395, 1975.
- [8] 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.
- [9] 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.
- [10] R. D. Blumofe and C. E. Leiserson, “Scheduling multithreaded computations by work stealing,” *J. ACM*, vol. 46, no. 5, pp. 720–748, 1999.
- [11] J.-N. Quintin and F. Wagner, “Hierarchical work-stealing,” in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, ser. EuroPar’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 217–229. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1887695.1887719>

Analysis of GPGPU Platforms Efficiency in General-Purpose Computations

P. Kartashev¹ and V. Nazaruk¹

¹Institute of Applied Computer Systems, Riga Technical University, Riga, Latvia

Abstract - Nowadays a technique of using graphics processing units (GPUs) for general-purpose computing (or GPGPU) is becoming more and more widespread. The goal of this paper is to analyze efficiency of computing with use of the GPGPU technique, depending on several factors. In this paper, there are analyzed differences in performance and platform organization between widespread GPGPU computational platforms (both hardware and software). There are also described differences between CPU and GPU computations, as well as presented performance measurements for some GPGPU hardware architectures. This paper can help software developers choose more appropriate ways to implement specific fairly large computational tasks.

Keywords: graphics processing units (GPUs), general-purpose computing on graphics processing units (GPGPU), OpenCL

1 Introduction

Many modern computers (approximately since 2006) have video cards that can be used not only for performing calculations connected with graphics, but also for arbitrary (even not related with graphics) calculations. Such technique of using graphic processing units for general-purpose calculations is called *general-purpose computing on graphics processing units (GPGPU)*.

Therefore, nowadays (in contrast to a period of several years before) most processing systems belong to one of the following two classes:

- central processing units (CPUs),
- graphics processing units (GPUs).

In order to use GPUs for computations, it is needed to write a program which uses a specific GPGPU programming model and architecture. Nowadays there exist several GPGPU platforms, which implement some different programming models and/or architectures; most notable of them include NVIDIA CUDA, OpenCL, Microsoft DirectCompute, ATI Stream.

For some kind of applications (usually for those which are multi-threaded and/or parallel), the use of general-purpose computations on modern GPUs can achieve speeds way beyond that on modern CPUs. Therefore, the use of graphics processing units for general-purpose computations is a topical

sphere of research nowadays. The goal of this paper is to analyze efficiency of computing with use of GPGPU technique, depending on several factors, including target processing units, as well as GPGPU platforms themselves.

When speaking about the efficiency of GPGPU platforms, the thing that should be considered first is execution speed of programs which use the GPGPU technique. This mostly depends on specific processing units used for calculations, but also on a specific GPGPU platform architecture and programming model.

In this paper, there are analyzed and explained differences in performance and platform organization between GPGPU computational platforms (both hardware and software). Such GPGPU model comparison can help developers choose from these platforms to achieve best compatibility, speed, and portability for their GPGPU applications. Some guidelines for GPGPU developers, when they can use each of these platforms best, are formulated.

In the paper, there are also described differences between CPU and GPU computations. In our work, a comparison of GPU and CPU instructions is provided. There are presented performance measurements for GPGPU hardware architectures (including information about performance and time utilization of target processing units); some of the advantages and disadvantages of platforms are determined. Results concerning the performance measurements are based on practical experiments: by the authors, there was written and utilized an application (for the OpenCL programming model) for measuring the time of execution of different types of calculations. The methodology used is discussed further in this paper.

2 General-purpose computations on GPUs

A GPU is specialized for compute-intensive, highly parallel computation — exactly what graphics rendering does — and therefore designed in such a way that more transistors are devoted to data processing rather than data caching and flow control. A GPU is suited to problems that can be expressed as data-parallel computations — the same program is executed on many data elements in parallel — with high arithmetic intensity. Same program is executed for each data element, there is a lower requirement for sophisticated flow

control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D graphics, rendering large arrays of pixels and vertexes are processed in parallel are applied to parallel threads [1]. Unlike CPUs, GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very fast. This approach of solving general purpose problems on GPUs is known as GPGPU. GPU has advantage over CPU by running data in parallel — benefit many tasks such as video/audio processing, large data sets processing, computational modelling (as industrial, weather, nature, particle simulation), ray-tracing, post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition. Many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology [1].

The main GPU advantage over CPU is its high throughput. Whilst CPU performance now increases only ~26% a year, GPU performance increases more than 100% a year.

GPU uses different architecture using many ALU units in the chip is the main difference from CPU, for example AMD PHENOM II X4 has 12 ALU, but GeForce GT240 GPU — 96 ALU (see Table 1).

GPU developers provide free GPU programming libraries (or SDKs), e. g. OpenCL, CUDA by Nvidia, Stream SDK by AMD.

CUDA (an acronym for “Compute Unified Device Architecture”) is a parallel computing architecture developed by NVIDIA. CUDA is the computing engine in NVIDIA graphics processing units (GPUs) that is accessible to software developers through variants of industry standard programming languages. CUDA is accessible to software developers through C for CUDA, CUDA Fortran Compiler and third party language wrappers, such as Jcuda, pyCUDA, etc. CUDA has been used to accelerate non-graphical applications. Programmers use C for CUDA (C with NVIDIA extensions and certain restrictions), compiled through NVCC compiler to code algorithms for execution on the GPU. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.

CUDA uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions. However, a single process must spread across multiple disjoint memory spaces, unlike other C language runtime environments. Fermi GPUs now have (nearly) full support of C++ [2].

OpenCL (Open Computing Language) is a framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. OpenCL provides parallel computing using task-based and data-based parallelism.

OpenCL includes a language based on the C99 standard for writing kernels, plus APIs that are used to define and then control the GPGPU platforms. Programs written on OpenCL can access GPU of all supported GPU vendors for GPGPU computations. The OpenCL specification is under development by Khronos Consortium, which is open to everyone [3].

Microsoft’s DirectCompute is a new GPU Computing API that runs under both Windows Vista and Windows 7. DirectCompute is supported on current DirectX 10 class GPUs, DirectX 11 GPUs. It allows developers to harness the massive parallel computing power of GPUs to create compelling computing applications in consumer and professional markets [4].

GPGPU platform comparison is described in further sections.

Table 1. Comparison of modern CPU and GPU used for the measurements in this paper

	AMD Phenom II X4	NVIDIA GeForce GT240
ALU in core / multiprocessor	3	8
Cores / Multiprocessors	4	12
Total ALU	12	96
Peak theoretical performance, GFLOPS	48 (3000 MHz)	385.9 (MADD+MUL) (3 instructions per cycle) (~1400 MHz)

3 Related works

In this section, there are discussed similar works to the research topic.

V. Volkov’s work [5] shows that matrix manipulation with GPU can achieve speedup up to 2 times greater in double precision and 4–8 times for single precision than CPU. Comparison of performance is maintained. In [6], the authors provide a research study to achieve 10^3 speedup by using algorithm implementation with CUDA. Some works implement whole complex of algorithms on GPU: for example, the work [7] shows up to 20–100 times speedup, by implementing SQL-Lite SQL engine on CUDA architecture. The work [8] shows 20x speedup over CPU in AES cryptography. Research has been done by using NVIDIA CUDA. The work [9] proves that GPGPU computing problem is high PCI-E latency and low bandwidth, and sometimes optimizations required to achieve performance and there is big speedup in processing when using large data-set processing on GPU.

These works prove the efficiency of GPU based algorithms and describe useful uses of GPU. GPU has been used

mainly for scientific computations and large data processing. In this field, according to preceding works, GPU mostly outperforms CPU.

4 Comparison of GPGPU platforms

In this section, most widespread GPGPU programming models are described in context of comparison with each other.

We describe native programming language support and well as support for third party languages, for example Java, Python. We compare 3 main GPGPU platforms: OpenCL, CUDA, DirectCompute. We provide 3 criteria for the comparison:

- 1) Portability: what operating systems GPGPU computations could be made on?
- 2) Third party language support: is there support for GPGPU platform function call from other programming languages?
- 3) Possible execution on CPU (heterogeneous computing).

The weakness of DirectCompute is that it uses compute shader, which has specific restrictions: initialize a Direct3D device, create data buffers (resources) for shader, set shader state and launch it. Specific programming rules must be met.

As one can see, the DirectCompute API relies on DirectX 10 or 11 API to initialize GPU and make computations possible.

Despite DirectCompute benefits, that there is no need to special driver to make GPGPU computations using DirectCompute, and computations will run on every GPU that supports DirectX 10 or 11. DirectCompute is available for Windows Vista/7 only.

OpenCL and CUDA provide more flexibility and easy-to-write applications for GPU. We do not need compute shader to write and execute CUDA and OpenCL application. CUDA is available to only NVIDIA GPU's – application which was written for NVIDIA CUDA platform cannot be executed on ATI GPU's. OpenCL in comparison can be executed on various kinds or processing units, the only request is OpenCL driver from manufacturer of processing unit.

For the comparison summary, see Table 2.

Table 2. Summary of GPGPU programming model comparison

	OpenCL	CUDA	DirectCompute
Programming	C/C++ extensions	C/C++ extensions	C/C++, Shader Language
Portability	Windows, Linux, MacOS	Windows, Linux, MacOS	Windows Vista/7 with DirectX 10/11
API	OpenCL API	CUDA API	DirectX 11 API

	OpenCL	CUDA	DirectCompute
Third party language support	yes (JOCL, PyOPENCL etc.)	yes (JCUDA, pyCUDA, Fortran PGI CUDA compiler etc.)	no
Heterogeneous computing possible?	yes	partial (only with program recompilation)	no (possible execution only on GPU)

5 Analysis of an impact of a GPGPU platform on computations

As it is stated before in this paper, use of GPGPU in an application can have an effect on the resulting characteristics of computations. The impact can be made, for example, by a target architecture for a GPGPU application. Such an impact is analyzed next; as well as further in this section, there is analyzed a possible impact of a platform on the performance of a GPGPU application.

5.1 Comparison of possible target hardware architectures for GPGPU: CPUs and GPUs

Platforms for general-purpose computing on graphics processing units (for example, OpenCL and CUDA) provide ways to execute an application written with a GPGPU technique also on computers where there are no GPGPU-compatible GPUs. In these cases, all instructions of the programs are executed on CPU — a GPGPU environment is imitated on CPU in a way that is transparent for an executing program.

This means that there exist two main target processing unit models for programs with GPGPU: when a program is executed both on a CPU and a GPGPU-enabled GPU, and when it is run only on a CPU. Therefore, it is important to compare which each other these two possible modes of operating for a GPGPU program.

For much software, the speed of their execution is of great importance. Bottlenecks for this speed usually are a processor (or processors) on which the software is executed, as well as memory and buses. However, when the software highly depends on calculations, or in the software there are many continuous uniform operations, the execution speed is mostly dependent on performance of the processing units.

In order to efficiently maximize the speed of execution of an application, a processing unit should be used to the extent possible.

All GPUs suitable for arbitrary calculations (i. e., with a support of GPGPU) are multi-core (for example, NVIDIA GeForce 580 GTX consists of 16 multiprocessors); and a large number of modern CPUs are also multi-core.

Modern GPUs, in contrast to CPUs, are composed of a large number of cores. Moreover, computational power of GPUs in average is comparable to (and mostly larger than) computational power of CPUs. This means that GPUs (as well as multi-core CPUs) provide a big possibility for speeding up execution of applications [10].

A majority of common algorithms are defined in a sequential way (i. e., the corresponding code of an algorithm is sequential). However, the fact that nowadays most of modern processors are multi-core assumes that for a specific sequential algorithm in order to execute efficiently, it should be parallelized — divided into several maximally independent (parallel) tasks. Thus, in order to take advantage of using multi-core processors, algorithms should be adapted for parallel execution [10].

Despite both CPUs and GPUs are multi-core, their architectures differ significantly. According to Flynn's taxonomy [11], multi-core CPUs have in general a *MIMD* (Multiple Instruction stream, Multiple Data stream) architecture, with each core usually having a support for a set of SIMD (Single Instruction, Multiple Data) instruction. Alternatively, all GPGPUs have a *SIMD* architecture [10].

The difference between the architectures of multi-core CPUs and GPGPUs substantiates differences between optimization processes for these two types of processing systems. However, all optimizations for a SIMD architecture are also applicable to a MIMD architecture — because a MIMD architecture can be considered as a more enriched SIMD architecture [10]. This means that when writing an application for a heterogeneous GPGPU programming model and targeting and correspondingly optimizing it for execution on a SIMD GPU, the optimizations will work and will have effect also when executing on a CPU.

There are differences between x86 CPU instructions and GPU instructions — GPU takes with one instruction also memory reference (address). This makes addressing more effective. Also GPU can deploy single instructions with many operands into SIMD array, which can consist of 8–512 (NVIDIA GPU) ALU. This makes developing parallel applications in a more effective way. In CUDA, OpenCL, DirectCompute, there is an emended native parallelism support. That make sense for example GPU executes parallel code 100 times faster than CPU, but CPU executes serial code 50 times faster than GPU. It is efficient to combine CPU and GPU to make possible heterogeneous computing with task divergence.

5.2 Analysis of performance of GPGPU applications

If one wants to use a processing unit to the maximal extent, before implementing an application it is good to know some guidelines, what actions which will perform the application are supposed to be fast, and which actions are supposed

to be slow while running on a specific processing unit. In case of slow actions, at the application design stage, it is useful to avoid using slow operations to the extent possible. Therefore, it is useful to know, which operations on a specific application platform will perform faster, and which — slower.

In this section there are described practical results concerning speed of execution of primitive unary and binary operations (including basic arithmetical and bitwise Boolean operations) for three commonly used data types (*char*, *int*, and *float*) on the OpenCL GPGPU platform. OpenCL as a programming model was chosen mainly because of its support for ATI, NVIDIA GPUs and CPU, as well as multiple operating systems. OpenCL provides opportunity to run the same code on CPU and GPU.

For the measurement of speed, by the authors there was written a test application — an OpenCL program in the C++ programming language. To maximally smooth out the measurement errors, to measure small amounts of time with a high precision, each operation with the same input data was called 10 million times. This was implemented in a following way: an OpenCL kernel contained one operation (or a block of several similar operations), and the kernel was executed a specific number of times (for different data values) in a special loop (provided by an OpenCL programming model). The measurement of time intervals needed for the kernel to execute was implemented in the following way: the system time was measured just before and just after the execution of the kernel, and the difference of these values was considered as the execution time. The system time was measured using system calls, with the precision of several milliseconds.

The test programs (32-bit) were executed on a computer with an AMD Phenom II X4 965 CPU (3.40 GHz in each of 4 cores), 4 GB RAM, and 64-bit Microsoft Windows 7 operating system, and the following video adapters:

- GPU ATI = ATI Radeon HD 5750,
- GPU NVIDIA = NVIDIA GeForce 240 GT.

With GPU NVIDIA due to technical problems there were measured only operations on the *char* data type.

It is needed to be stated that the obtained CPU performance is when forcing to run an OpenCL application on a CPU, not GPU. This means that in such a way obtained performance is not the same as the performance of a CPU when an application is implemented especially for running on CPU (i. e., without a use of a GPGPU).

The generalized results of the experiments in different views are provided in Figure 1–Figure 3.

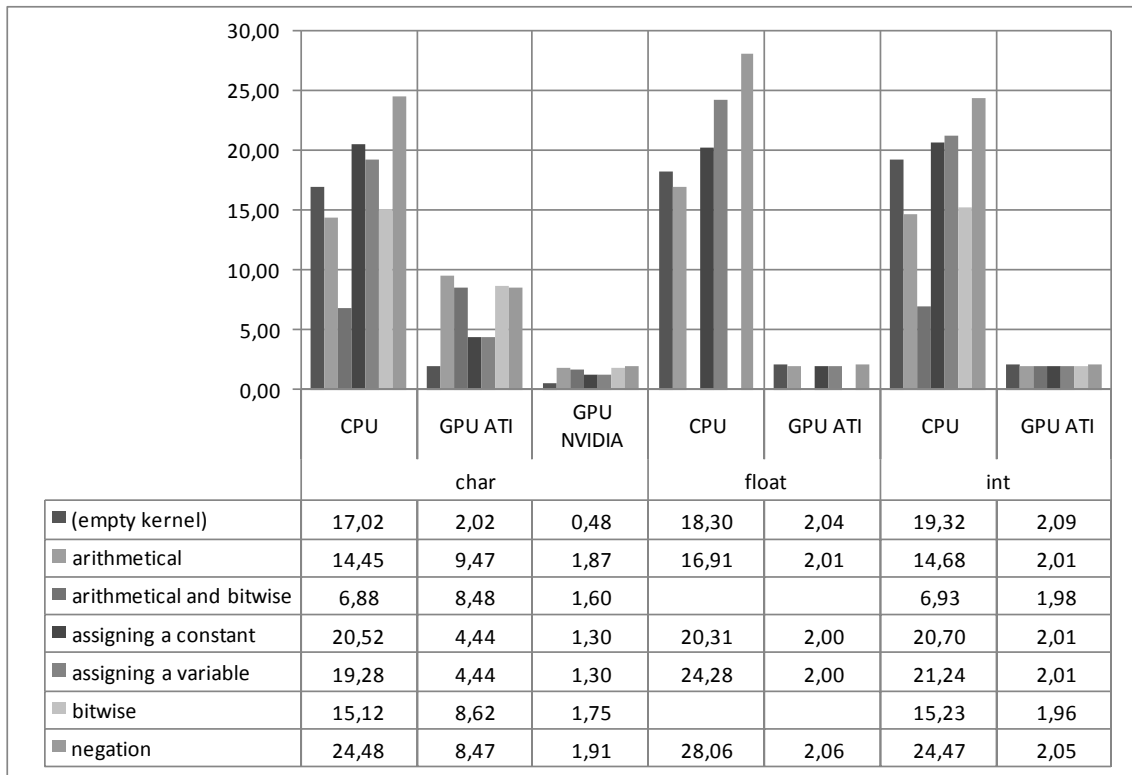


Figure 1. Time (in milliseconds) needed to execute 10 million equal primitive operations in an OpenCL kernel, depending on a type of operations, a data type, and a target processing unit

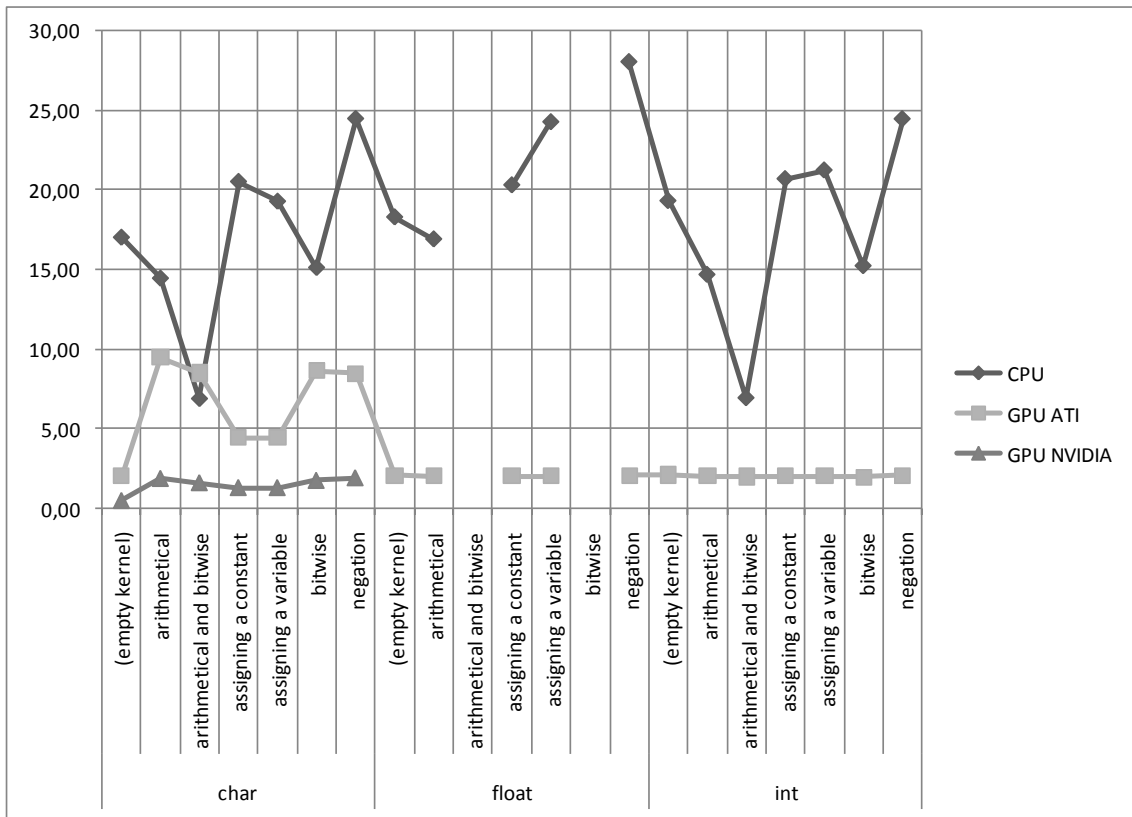


Figure 2. Time (in milliseconds) needed to execute 10 million equal primitive operations in an OpenCL kernel, depending on a type of operations, a data type, and a target processing unit. It is easy to see that almost in all tested cases the fastest target processing unit is NVIDIA GPU, and the lowest — CPU

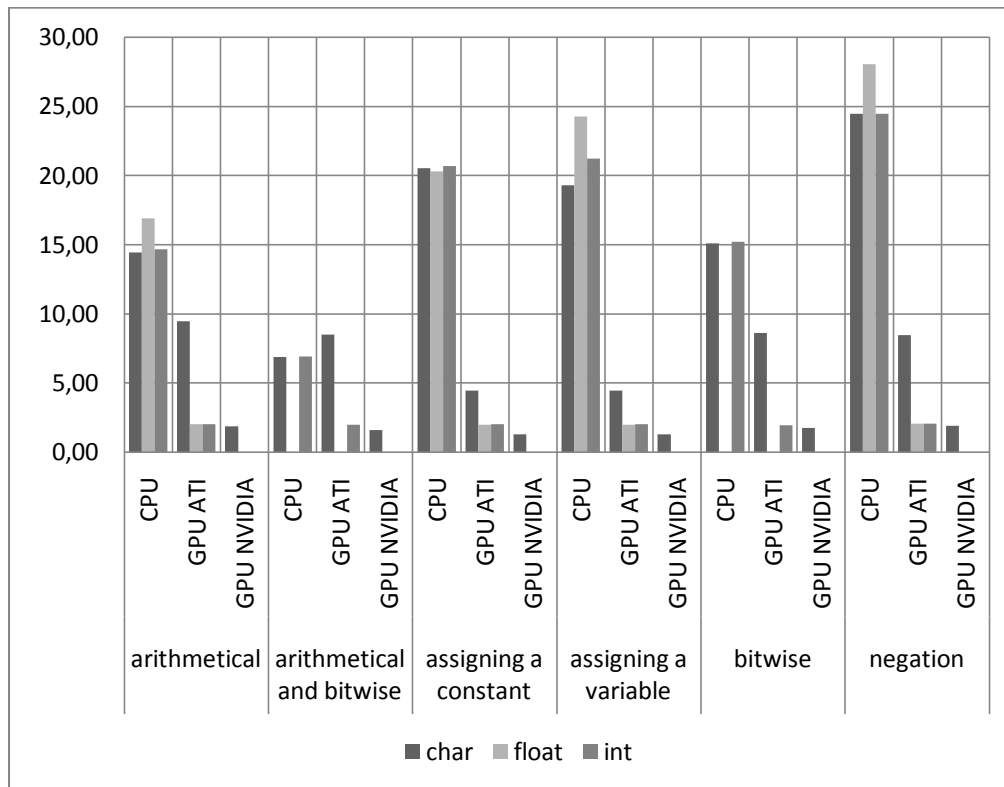


Figure 3. Time (in milliseconds) needed to execute 10 million equal primitive operations in an OpenCL kernel, depending on a type of operations, a data type, and a target processing unit. One can see that on GPUs all operations with *char* data type (which takes integer values from -128 to 127) are significantly slower than operations with *int* and *float* data types. However, on CPUs operations with *float* data type are approximately as fast as with *int* data type; the same situation is on CPUs with all data types

From Figure 2 one can see that the fastest target processing unit is NVIDIA GPU, and the lowest — CPU. This is true almost for all operations. The performance of two different GPUs should not be compared directly; however, comparing the performance of the GPUs with a performance of the CPUs, one can see that the formers are much higher than the latter.

From Figure 3 one can see that on GPUs all operations with *char* data type (takes integer values from -128 to 127) are significantly slower than operations with *int* and *float* data types. This situation is slightly different from the usual situation with CPUs, where operations with *char* operands perform much faster than operations with *int* and *float* operands. Also, from the figure it is seen that, on GPUs operations with *float* data type are approximately as fast as with *int* data type; the same situation is on CPUs with all data types.

The differences between the obtained results showing GPU instruction performance, and generally known results on the performance of CPU instructions (including that floating-point operations on a CPU are performed much slower than integer operations) can be explained with the differences in the instruction sets and architectures of GPUs and CPUs. (For example, in [12], there is described instruction set architecture for ATI Evergreen Family GPUs).

6 Conclusions

When designing programs for GPUs which support general-purpose computing, in order to make programs be efficient, it is necessary to be aware of some specific features of GPUs. This includes the knowledge of the performance level of primitive mathematic operations — as there was shown in this paper, in calculations, it is better not to use variables of small size (i. e., of *char/byte* data type) but replace them with *integer*-type or floating point variables.

When intending an application for a GPGPU platform, it is needed to be known that in case of there is no GPGPU-enabled GPU on a destination computer, the performance will somewhat suffer. Speaking about GPGPU software platforms, it can be stated that OpenCL and CUDA provide more flexibility and easiness to write applications for GPU: there is no need for a compute shader to be used to execute CUDA and OpenCL app (unlike in DirectCompute). However, DirectCompute benefits from the point of view that there is no need to special driver to make GPGPU computations using DirectCompute, and computations will run on every GPU that supports DirectX 10, 11 or later. In addition to the above, in many cases the use of a specific GPGPU software platform (except OpenCL) can be limited either by GPU manufacturer (CUDA supports only NVIDIA GPUs, and Stream supports only ATI GPUs), or by a running operating system (DirectCompute cannot be run, for example, on Linux).

One of the directions of the further work is to improve the application used for the experiments in order to be able obtain wider measurement results (for example, not only for arithmetic or Boolean operations). Also, the experiments should be made on a larger set of different processing units.

GPGPU computations benefit in such tasks as image processing, physics simulations, large array processing, and many others tasks which deals with large data sets. A significant importance is information synchronization between threads that uses shared memory. As CPU cores can also use shared memory, it is possible for the further research to compare CPU and GPU data synchronization. Also the future research may include a new field of study — heterogeneous computing, which include both CPU and GPU computations. This is topical when a CPU and a GPU are combined on single die. Future research in the heterogeneous computing and APU (accelerated heterogeneous processing units) field may give results in understanding how to accelerate today's algorithms and programs in order for them to run faster on heterogeneous processors. In further studies, there could be also discussed the way how computations can be transferred between CPU and GPU, and how effectively a program written for GPU can be translated for running on CPU, and vice versa.

7 References

- [1] "NVIDIA OpenCL Programming guide for the CUDA architecture, Version 3.2". [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/OpenCL_Programming_Guide.pdf. [Accessed: Oct 2010].
- [2] "CUDA - Wikipedia, the free encyclopedia". [Online]. Available: <http://en.wikipedia.org/w/index.php?title=CUDA&oldid=389564200>. [Accessed: Oct 2010].
- [3] "OpenCL - Wikipedia, the free encyclopedia". [Online]. Available: <http://en.wikipedia.org/w/index.php?title=OpenCL&oldid=389311710>. [Accessed: Oct 2010].
- [4] "DirectCompute - Wikipedia, the free encyclopedia". [Online]. Available: <http://en.wikipedia.org/w/index.php?title=DirectCompute&oldid=389634399>. [Accessed: Oct 2010].
- [5] V. Volkov and J. W. Demmel, "Benchmarking GPUs to tune dense linear algebra", University of California at Berkeley, SC08, November 2008. [Online]. Available: <http://parlab.eecs.berkeley.edu/sites/all/parlab/files/Benchmarking%20GPUs%20to%20tune%20dense%20linear%20algebra.pdf>. [Accessed: Oct 2010].
- [6] A. C. Thompson, C. J. Fluke, D. G. Barnes, and B. R. Barsdell, "Teraflop per second gravitational lensing ray-shooting using graphics processing units", Centre for Astrophysics and Supercomputing, Swinburne University of Technology, May 2009. [Online]. Available: <http://arxiv.org/pdf/0905.2453.pdf>. [Accessed: Oct 2010].
- [7] P. Bakkum and K. Skadron, "Accelerating SQL Database Operations on a GPU with CUDA", Department of Computer Science University of Virginia, GPGPU-3, March 2010. [Online]. Available: http://www.cs.virginia.edu/~skadron/Papers/bakkum_sqlite_gpgpu10.pdf. [Accessed: Oct 2010].
- [8] S. Manavski, "CUDA Compatible GPU as an Efficient Hardware Accelerator for AES Cryptography", 2007 IEEE International Conference on Signal Processing and Communications (ICSPC 2007), 24–27 November 2007, Dubai, United Arab Emirates: 2007. [Online]. Available: <http://www.manavski.com/downloads/PID505889.pdf>. [Accessed: Oct 2010].
- [9] R. V. van Nieuwpoort, J. W. Romein, "Using Many-Core Hardware to Correlate Radio Astronomy Signals", Netherlands Institute for Radio Astronomy, 23rd ACM International Conference on Supercomputing. [Online]. Available: <http://www.cs.vu.nl/~rob/papers/ics09-correlator.pdf>. [Accessed: Oct 2010].
- [10] V. Nazaruk and P. Rusakov, "Implementation of Cryptographic Algorithms in Software: An Analysis of the Effectiveness", Scientific Journal of Riga Technical University, Vol. 43, pp. 97–105, 2010.
- [11] M. Flynn, "Some Computer Organizations and Their Effectiveness". IEEE Transactions on Computers, Vol. C-21, Issue 9, pp. 948–960, 1972.
- [12] "Evergreen Family Instruction Set Architecture. Instructions and Microcode. Reference Guide", Advanced Micro Devices, Inc., September 2010. [Online]. Available: http://developer.amd.com/gpu/ATIStreamSDK/assets/AMD_Evergreen-Family_Instruction_Set_Architecture.pdf. [Accessed: Oct 2010].

Study of Performance Issues on a SMP-NUMA System using the Roofline Model

Juan A. Lorenzo¹, Juan C. Pichel¹, Tomás F. Pena¹, Marcos Suárez² and Francisco F. Rivera¹

¹Computer Architecture Group, Electronics and Computer Science Dept., Univ. of Santiago de Compostela, Santiago de Compostela, Spain.

²Land Laboratory, Univ. of Santiago de Compostela, Lugo, Spain.

Abstract—This work presents a performance model based on a combination of hardware counters and a Roofline Model developed to characterize the behavior of the FINISTERRAE supercomputer, one of the largest SMP-NUMA systems. Our main objective is to provide an insightful model which allows to determine, at a glance, performance issues related to thread and memory allocation of irregular codes in this machine. Results show that this model provides practical information of the effects that degrade the performance of a code and, additionally, gives hints to improve it.

Keywords: Roofline Model, Irregular Codes, Hardware Counters.

1. Introduction

State-of-the-art architectures involve many cache levels in complex several-node NUMA configurations with different number of multi-core processors. A good example is the supercomputer FINISTERRAE installed at the *Galicia Supercomputing Center* in Spain [1]. FINISTERRAE is a SMP-NUMA system with more than 2500 Itanium2 Montvale processors and 19 TB of RAM memory. Designed to undertake great technological and scientific computational challenges, it is one of the biggest shared-memory supercomputers in the World. Improving the performance and scalability of dense or sparse codes on such multicore architectures can be extremely non-intuitive. Although there exist some stochastic analytical [2] models and statistical performance models [3] which can accurately predict performance, they rarely provide insight into how to improve the performance of programs, compilers and computers. We approached this problem by developing a Roofline Model for FINISTERRAE.

The Roofline Model [4] provides realistic expectations of performance and productivity. It does not try to predict program performance accurately. Instead, it integrates in-core performance, memory bandwidth and locality into a single readily understandable performance figure, showing inherent hardware limitations for a given computational kernel, potential benefit and priority of optimisations. In this regard, this work demonstrates that hardware counters can assist in this task.

Subsequent sections will introduce the Roofline Model developed for FINISTERRAE and the main results obtained.

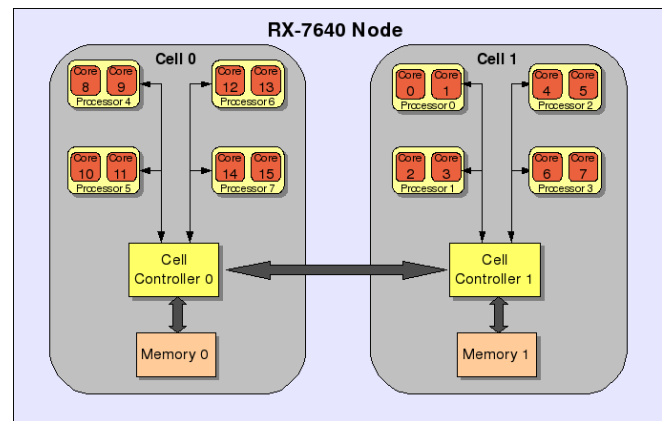


Fig. 1: HP Integrity Rx7640 node.

2. FINISTERRAE's Roofline Model

The Roofline Model relies on three metrics: *Computation* measured as GFlops/s, *Communication* measured as DRAM bandwidth (GBytes/s) and *Locality*. The metric that relates performance to bandwidth is defined as *Operational Intensity*. It is measured in *Flops/Byte* and means “FP operations performed per byte of DRAM traffic transferred”. That is, traffic is measured between caches and memory, not between the processor and the caches. This measure predicts the DRAM bandwidth needed by a kernel on a particular computer. Figure 2 shows the roofs of our model for a Rx7640 FINISTERRAE node (see Figure 1). The plot is on log-log scale. The Y-axis shows the attainable double-point performance in GFlops/s. The X-axis displays the operational intensity. The horizontal line shows the peak floating-point performance of the computer, and its computation values can be derived either from the processor’s manual or from performance benchmarks. The maximum attainable bandwidth is a line of unit slope ($\frac{GFlops/s}{Flops/Byte} = GBytes/s$) and can be derived also from the architecture manual or from a benchmark. Both lines intersect at the point of peak computational performance and peak memory bandwidth. The peak floating-point performance as well as the maximum bandwidth of FINISTERRAE were obtained from the manufacturer’s documentation [5]. Note that the

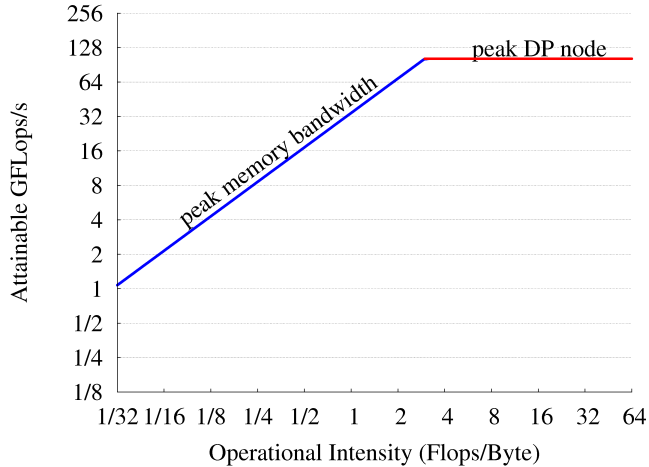


Fig. 2: Roofs of FINISTERRAE Roofline Model.

former (102.4 GFLOPs/s) comes from the product of a Montvale processor's peak performance times the number of processors in a FINISTERRAE node, whereas the latter (34.4 GBytes/s) comes from the maximum bandwidth of a memory bus times the number of buses in a node (see Figure 1). The attainable performance of a given kernel is upper bounded by both the peak flop rate, and the product of bandwidth and the flop:byte ratio.

$$Gflop/s = \min \left\{ \begin{array}{l} Peak\ Gflop/s \\ Peak\ Memory\ BW * actual\ flop : byte\ ratio \end{array} \right.$$

These roofs cannot ever be reached, since they are physical limits given by the architecture. They are defined once per multicore computer and can be reused for any kernel.

The next step consists in adding *ceilings* to the model. Ceilings are performance barriers which limit the maximum attainable performance. They suggest which optimisations to perform and the potential benefit to achieve. We cannot break through one ceiling without first performing the corresponding optimisation. Figure 3 depicts computational (horizontal lines) and bandwidth (slanted lines) ceilings. These ceilings are not fixed and can be chosen depending on the performance limits of interest. Since this study is focused on a FINISTERRAE node which comprises several processors, computational ceilings in the figure show performance limits depending on the number of cores involved. These values are derived from the architecture manual. Bandwidth ceilings are related to memory imbalance and were collected by a tuned version of the LMBench benchmark [6]. Each ceiling denotes the maximum sustained bandwidth attainable when all the traffic is concentrated in a single cell (i.e., is handled by a single memory controller) or when it is distributed between both cells. Ceilings, as well as roofs, are measured only once per multicore computer and can be reused for any kernel which runs in that machine.

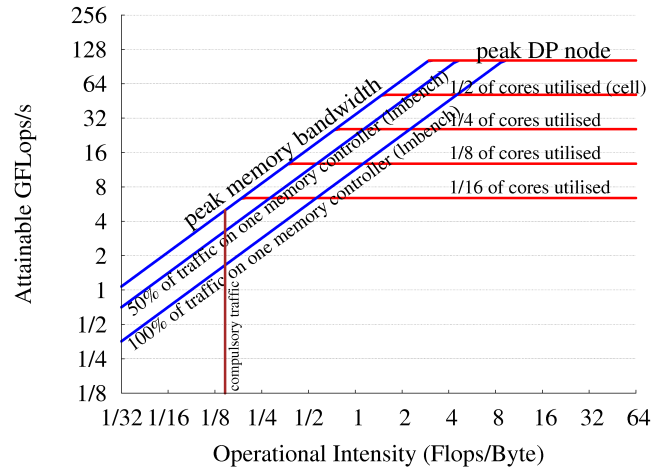


Fig. 3: Ceilings added to the FINISTERRAE Roofline Model and locality wall for a SpMV.

The last step to create the Roofline Model involves the computational kernel under study. The attainable performance for a kernel is related to its operational intensity. Indeed, moving to the right of the Roofline Model (towards higher values in the x-axis) means a higher number of FP operations per byte transferred from memory to cache and, therefore, a better performance. There is a limit in the maximum operational intensity of a kernel, given by the lower limit to communication: the compulsory traffic. This limit is called *locality wall* and is unique for a kernel-architecture combination. Note that the actual operational intensity may be lower due to cache misses. A kernel can be *cpu-bound* or *memory-bound* depending on whether its maximum operational intensity is on the right of the ridge point or on the left, respectively. Figure 3 shows the locality wall for a Sparse Matrix-Vector product (SpMV). This limit was obtained by reducing the size of the matrix A and the arrays x and y until they fit into cache, narrowing down the memory accesses to the compulsory traffic to fetch the data just once into cache. As seen in the figure, the SpMV is a memory-bound kernel, with a maximum operational intensity rather low due to its scarce reuse of the cache memory.

2.1 Experiment setup

Once sketched the roofs and ceilings of the Roofline Model, the next step consisted in measuring the performance of a target program to depict it according to the magnitudes of this model. In particular, our objective was to see graphically the influence of thread and data allocation in a way that could cast some light about which improvements might be made.

A parallel openMP SpMV with block distribution was used. To place a performance point in the Roofline Model, a pair of coordinates (*Operational Intensity*, *Attainable Per-*

formance]) are needed. In order to get them, PAPI [7] was used to instrument the code and access native events of the Montvale processor, measuring the following magnitudes:

- **Computation (GFlops/s):** The number of Flops of the kernel is given as the sum of the values per thread returned by the event `FP_OPS_RETIRED`. The elapsed time is given by the function `PAPI_get_real_usec()`. The Computation is calculated as the sum of all Flops divided by the time of the slowest thread.
- **Operational Intensity (Flops/byte):** The traffic between main memory and cache memory is measured in Montvale as the sum of all bus memory transactions, stated as the number of *full cache line transactions* (event `BUS_MEMORY_EQ_128BYTE_SELF`) plus the number of *less than full cache line transactions* (event `BUS_MEMORY_LT_128BYTE_SELF`). The number of bytes transferred per thread is then calculated according to the formula: $Bytes\ transferred = BUS_MEMORY_EQ_128BYTE_SELF * 128 + BUS_MEMORY_LT_128BYTE_SELF * 64$. The whole number of bytes transferred by the kernel is the sum of the number of bytes per thread. The value of the operational intensity is calculated as the quotient between the sum of Flops from all threads and the sum of bytes transferred by all threads.

A set of matrices from the *University of Florida Sparse Matrix Collection* (UFL) [8] was represented using the Roofline Model. Two of these matrices, `pct20stif` and `exdata_1`, chosen as being paradigmatic of a well and a badly-balanced matrix, respectively, are analyzed here. Table 1 shows their imbalance as a result of dividing each matrix in blocks for 2, 4, 8 and 16 threads. This value is the quotient between the blocks with lowest and highest NNZ values. The closer to 1, the better the balance. It seems clear, therefore, how `pct20stif` is much better balanced than `exdata_1`.

2.2 Experiment #1: Default parallelization

In this configuration, the parallel SpMV was executed for 1, 2, 4, 8 and 16 threads. The Linux scheduler was allowed to map threads to cores at its will. Data were allocated by the default system first-touch policy. Figure 5 shows the performance of the parallel SpMV for matrices `pct20stif` (a) and `exdata_1` (b). Each red cross displays the performance of each n-thread case. Note that matrix `pct20stif` shows a much more regular pattern than `exdata` (see Figure 4), which concentrates most of its nonzero values into a small region. Some interesting information can be inferred from

	2p	4p	8p	16p
<code>pct20stif</code>	0,9386	0,8816	0,8008	0,703
<code>exdata_1</code>	0,0046	0,002	0,002	0,002

Table 1: Matrix imbalance of `pct20stif` and `exdata_1`.

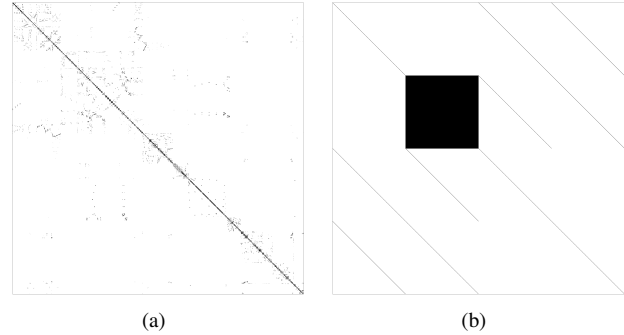


Fig. 4: Matrices `pct20stif` (a) and `exdata_1` (b).

their Roofline Models: Figure 5(a) shows performance points that grow both in computation and operational intensity as the number of processors increases. Indeed, given the big size of the matrix, the 1-thread case shows a high number of conflict misses, which reduces the ratio flops:byte (i.e. the operational intensity) with respect to the maximum value (the compulsory misses wall). Consequently, the number of GFlops/s is not as high as it could be if the conflict misses were lower. As the number of threads increases, the matrix (and the arrays x and y) is shared out among the processors. Therefore, the number of conflict misses decreases, the points in the graph shift towards the compulsory wall, and the computation value increases.

Note that, whereas the number of processors duplicates in each case, the points in the graph are not equidistant. There is a bigger gap between 2 and 4 processors, and between 8 and 16, than the remaining cases. Quantified, the ratio of attainable GFlops/s between 2 and 1 thread is $\sim 1,35$. So it is with the ratio between 8 and 4 threads. However, the ratio between the 4 and 2 threads is $\sim 1,9$, and between 16 and 8 threads is $\sim 2,7$. Two reasons for this can be found here. Firstly, the number of conflict misses does not decrease linearly as the number of processors increases. Thus, the shift increment of each point towards the compulsory wall is not constant. Secondly and more important, the SpMV considered uses its master thread to allocate all data before starting the computation. Therefore, the system's first touch

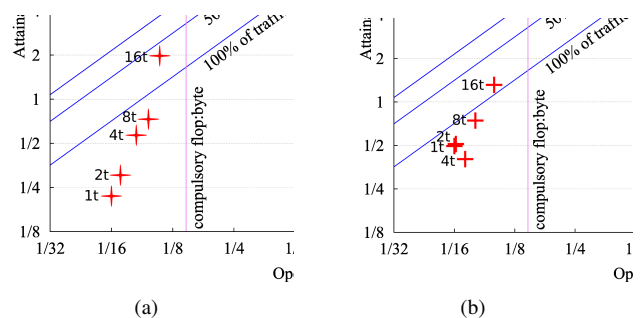


Fig. 5: SpMV results for `pct20stif` (a) and `exdata_1` (b).

policy will place all data uniquely in the master's memory.

When using 2 threads, they are bound to have been attached to cores in different cells. One of them will need to fetch data remotely and this explains the little difference in performance between 1 and 2 threads. However, for the 4-thread case, the scheduler has mapped most of the threads to cores in the same cell and, therefore, the performance is much higher. Again, for a 8-thread case, the difference in performance with the 4-thread case is not too noticeable, which implies that threads have been spread out between both cells. For 16 threads all cores are used, although 8 of them fetch data from a remote cell. Therefore, performance will be higher than for a 8-thread case, but not as high as the architecture allows (the 16-thread point is far from the peak memory bandwidth).

Note that, while `pct20stif` is a matrix with a pattern with diagonal shape, `exdata_1` has most of its data concentrated in a small region. Hence, its pattern is prone to cause load balance problems. The representation of its performance in the Roofline Model (shown in Figure 5(b)) substantially differs from the one of `pct20stif`. The performance achieved is virtually identical for 1 and 2-thread cases. Indeed, openMP divides evenly the number of rows among the available threads. Splitting this matrix in two halves (in terms of number of rows) gives $\sim 99.6\%$ of the load to one single thread in the 2-thread case. Therefore, a glance to the figure attracts our attention to an important load imbalance problem.

The 4-thread point for `exdata_1` is placed below the points for 1 and 2 threads and slightly to the right of them. This means that, whereas this case yields a worse performance, its operational intensity is better. Therefore, the load is badly balanced but the bus bandwidth is less saturated than in previous cases. So we can conclude that the scheduler spreads out two threads to each cell, but keeping most of the load in the same cell. Finally, 8 and 16 threads provide a finer distribution of the matrix among the threads in both cells, so the performance increases in both GFlops/s and operational intensity.

2.3 Experiment #2: Exploiting thread allocation

The SpMV was run with 1, 2, 4, 8 and 16 threads. Threads were mapped to cores explicitly and data were allocated manually to memory modules using the Linux `numactl` command. Whereas in the previous section all threads had been mapped to cores by the Linux scheduler and data had been allocated in cell 0 due to the system's first-touch policy, in this experiment we made sure that threads 1 to 8 were mapped to cores in cell 0, as well as their data. Specifically, the threads were mapped alternately to cores in both buses of a same cell. Note that all threads were placed as far as possible from the remaining ones in order to spread the

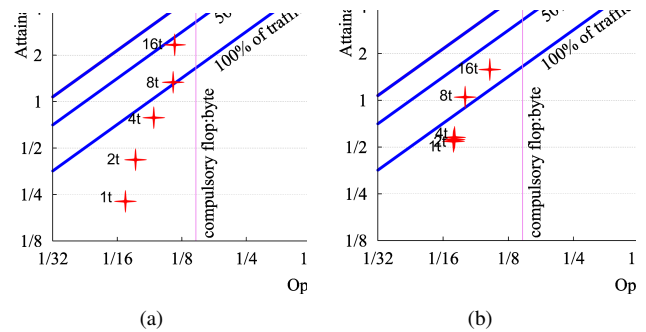


Fig. 6: Exploiting thread distribution of SpMV for matrices `pct20stif` (a) and `exdata_1` (b).

threads out among the available buses. For 16 threads, data were allocated in the interleaving zone.

Figure 6 shows the outcomes for matrices `pct20stif` and `exdata_1`. Note the distribution of points for cases 1p to 8p in Figure 6(a). In this case all the performance values are vertically equidistant from each other. Since `pct20stif` is a well-balanced matrix, the only remaining cause for imbalance would be the thread allocation. However, in this case threads have been manually mapped to cores in the same cell where data are, taking care of keeping them always balanced between both buses in the cell. That justifies the well-distributed points in the Roofline Model.

16 threads is a particular case. We noticed an increase in performance as steady as in previous cases. However, the operational intensity is almost alike. In this case, a decrease in performance was expected because data were allocated in the interleaving memory, which presents a latency higher than local data. A rough calculation showed that the memory needed to allocate the matrix and the arrays x and y is 71, 64 Mbytes. This size, divided by either 8 or 16 processors, yields a value below the 9 MBytes size of the L3 cache memory. Therefore, although the higher latency will influence the performance, only the compulsory cache misses together with a limited amount of conflict misses (which prevents the performance points to reach the compulsory wall) will occur. That is the reason why the operational intensity is similar in both cases. Figure 6(b) shows the Roofline Model for matrix `exdata_1`. As expected, the only difference in performance with Figure 5(b) is the 4-thread case. The manual thread allocation in cell 0 balances better the data between the buses in the cell. However, the imbalance due to the irregular nature of the matrix still exists, and that is why the performance is practically the same previously obtained for 1, 2 and 4 threads.

3. Conclusions

This work has presented the development of a *Roofline Model* for a node of the FINISTERRAE supercomputer. The experiments performed using a parallel SpMV stated that codes with a high level of cache replacement and data size larger than the L3 cache should have their threads shared out among the available buses and their data placed in the same cell. The Roofline Model has provided us with an insightful way to provide information at a glance about the performance of a program related to load balance and both thread and data allocation issues.

4. Acknowledgements

This work has been partially supported by Hewlett-Packard under contract 2008/CE377, by the Ministry of Education and Science of Spain, FEDER funds under contract TIN 2010-17541 and by the Xunta de Galicia (Spain) under contract 2010/28 and project 09TIC002CT. The authors also wish to thank the supercomputer facilities provided by CESGA.

References

- [1] *Galicia Supercomputing Center*, <http://www.cesga.es>.
- [2] A. Thomasian and P. Bay, "Analytic queueing network models for parallel processing of task systems," *Computers, IEEE Transactions on*, vol. C-35, no. 12, pp. 1045–1054, dec. 1986.
- [3] E. Boyd, W. Azeem, H.-H. Lee, T.-P. Shih, S.-H. Hung, and E. Davidson, "A hierarchical approach to modeling and improving the performance of scientific applications on the ksr1," in *Parallel Processing, 1994. ICPP 1994. International Conference on*, vol. 3, aug. 1994, pp. 188–192.
- [4] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, April 2009.
- [5] *HP Integrity rx7640 Server Quick Specs*, http://h18000.www1.hp.com/products/quickspecs/12470_div/12470_div.pdf.
- [6] L. McVoy and C. Staelin, "LMbench: portable tools for performance analysis," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23.
- [7] *Performance Application Programming Interface (PAPI)*, <http://icl.cs.utk.edu/papi/>.
- [8] T. A. Davis, "The University of Florida sparse matrix collection," *NA DIGEST*, 1997.

Performance Modeling of Intel and Portland Compilers Using Westmere-Based Infiniband HPC Cluster

Muhammed Al-Mulhem and Raed Al-Shaikh
Department of Computer Science
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia
 {mulhem, g199607190}@kfupm.edu.sa

Abstract - In recent years, we have witnessed a growing interest in optimizing the parallel and distributed computing solutions using scaled-out hardware designs and scalable parallel programming paradigms. This interest is driven by the fact that the microchip technology is gradually reaching its physical limitations in terms of heat dissipation and power consumption. Therefore and as an extension to Moore's law, recent trends in high performance and grid computing have shown that future increases in performance can only be reached through increases in systems scale using a larger number of components, supported by scalable parallel programming models. In this paper, we evaluate the performance of two commonly used parallel compilers, Intel and Portland's PGI, using a state-of-the-art Intel Westmere-based HPC cluster. The performance evaluation is based on two sets of experiments, once evaluating the compilers' performance using an MPI-based code, and another using OpenMP. Our results show that, for scientific applications that are matrices-dependant, the MPI and OpenMP features of the Intel compiler supersede PGI when using the defined HPC cluster.

Index Terms— HPC, Intel, PGI, compilers, Infiniband.

I. INTRODUCTION

In recent years, we have witnessed a growing interest in optimizing the parallel and distributed computing solutions using scaled-out hardware designs and scalable parallel programming paradigms. This interest is driven by the fact that single CPU-chips are reaching their physical limits in terms of heat dissipation and power consumption. Therefore and as a continuation to Moore's law, recent trends in high performance and grid computing have shown that future increases in performance can only be achieved through increases in systems scale using a larger number of components, which are supported by scalable parallel programming models. Accordingly, scaled-out computing is clearly becoming the trend.

In terms of the underlying hardware, multi-cores CPUs and ultra-fast interconnects are today's ingredients for the High Performance Computing systems. Intel and AMD are still the leaders in the CPU industry, dominating the top500.org list of the most powerful supercomputers worldwide, and taking over 80% of HPC as of 2010 [9]. Nowadays, most of the high performance clusters use multi-

core CPUs in their compute nodes, ranging from 2 to 4 cores per nodes, while 6-cores sockets will become more common on clusters as Intel and AMD released their Westmere and Phenom II multi-core CPUs, respectively [7]. On the HPC interconnects side, there are several network interconnects that provide ultra-low latency (less than 1 microsecond) and high bandwidth (several gigabytes per second). Some of these interconnects may even provide flexibility by permitting user-level access to the network interface cards for performing communication, and also supporting access to remote processes' memory address spaces [1]. Examples of these interconnects are Myrinet from Myricom, Quadrics and Infiniband [1]. The experiments in this paper are done on the Infiniband architecture, which is one of the latest industry standards, offering low latency and high bandwidth as well as many advanced features such as Remote Direct Memory Access (RDMA), atomic operations, multicast and QoS [2]. Currently, available Infiniband products can achieve latency of 200 nanoseconds for small messages and a bandwidth of up to 3-4 GB/s [1]. As a result, it is becoming increasingly popular as a high-speed interconnect technology option for building high performance clusters.

On the parallel programming level, MPI and OpenMP have become the de facto standard to express parallelism in a program. OpenMP provides a fork-and-join execution model, in which a program begins execution as a single process or thread. This thread executes sequentially until a parallelization directive for a parallel region is found. At this time, the thread creates a team of threads and becomes the master thread of the new team. All threads execute the statements until the end of the parallel region. Work-sharing directives are provided to divide the execution of the enclosed code region among the threads. The advantage of OpenMP is that an existing code can be easily parallelized by placing OpenMP directives around time consuming loops which do not contain data dependences, leaving the source code unchanged. The disadvantage is that it is a big challenge to scale OpenMP codes to tens or hundreds of processors. One of the difficulties is a result of limited parallelism that can be exploited on a single level of loop nest.

Another program parallelization can be achieved through the message passing programming paradigm, which can be

employed within and across several nodes. The Message Passing Interface (MPI) [4] is a widely accepted standard for writing message passing programs. MPI provides the user with a programming model where processes communicate with other processes by calling library routines to send and receive messages. The advantage of the MPI programming model is that the user has complete control over data distribution and process synchronization, permitting the optimization of data locality and workflow. The disadvantage is that existing sequential applications require a fair amount of restructuring for parallelization based on MPI.

Our objective in this paper is to evaluate the performance of two commonly used parallel compilers, Intel and Portland's PGI, using a state-of-the-art HPC cluster. As described in the evaluation section, the performance evaluation is based on two sets of experiments, once evaluating the compilers' performance using an MPI-based code (between cluster nodes), and another using OpenMP-based code (using a single cluster node with dual hexa-cores Westmere sockets). To the best of our knowledge, this is the first paper that discusses Intel and PGI compilers' performance based on the latest Intel's Westmere technology and Infiniband QDR interconnect.

The rest of the paper is organized as follows: In section 2, we briefly shed some light on the compilers, the Infiniband interconnect technology, the Intel Westmere CPU architecture, and the MPI implementations used to benchmark our compilers, while in section 3 we describe our experimental evaluation and interprets the benchmark results. We state our conclusion and future work in the last section.

II. BACKGROUND

In this section, we briefly describe the characteristics of both the Intel and PGI compilers. Also, we will shed light on the technologies used to benchmark the two compilers. These are: the Quad Data Rate (QDR) Infiniband interconnect technology, the Intel Westmere architecture, and the MPI implementations.

A. Intel and PGI Compilers

Both Intel C and Fortran compilers support compilation for IA-32, Intel 64, Itanium 2, processors and certain non-Intel but compatible processors, such as certain AMD processors [7]. The Intel compiler further supports both OpenMP 3.0 and automatic parallelization for SMP. With the add-on capability Cluster OpenMP, the compiler can also automatically generate MPI calls for distributed memory multiprocessing from OpenMP directives.

Similar to the Intel compilers, PGI C/C++ includes native parallelizing/optimizing OpenMP C++ and ANSI C compilers. In addition, PGI's server version includes the OpenMP and MPI parallel graphical debugger (PGDBG) and the OpenMP and MPI parallel graphical performance profiler (PGPROF) that can debug and profile up to 16 local

MPI processes. PGI Server also includes a precompiled MPICH message passing library.

Both Intel and Portland Group Inc. (PGI) continuously tune their compilers to optimize for hardware platforms to minimize stalls and to produce code that executes in the fewest number of cycles. Both compilers share many technical features and high-level optimizations, such as: interprocedural optimization (IPO), profile-guided optimization (PGO), and high-level optimizations (HLO) [7, 8]. High-level optimizations are optimizations performed on a version of the program that more closely represents the source code, such as loop interchange, loop unrolling, loop distribution and data-prefetch. These optimizations are usually very expensive and may take considerable compilation time.

Interprocedural optimization applies typical compiler optimization that may affect multiple procedures, multiple files, or the entire program. IPO aims to reduce or eliminate duplicate calculations, inefficient use of memory, and to simplify iterations such as loops. In addition, IPO reorders the procedures for better memory utilization and locality. IPO also incorporates typical compiler optimizations on the entire program, for example, removing codes that are never executed in a program.

Profile-guided optimization, on the other hand, refers to a mode of optimization where the compiler performs a sample run of the program across a representative input set. The data would then indicate which sections of the program are executed more frequently, and which areas are accessed less frequently. All optimizations benefit from profile-guided feedback because they are less reliant on heuristics when making compilation decisions.

B. Infiniband Architecture

Infiniband is a technology that provides a high bandwidth I/O communication over a high speed serial data bus. It uses a switched fabric topology, as opposed to a hierarchical switched network like Ethernet [2]. It is designed to directly route data from one point to another point through a switch, where all transmissions begin or end at a channel adapter (HCA). Each Infiniband processor contains a host channel adapter (HCA) and each peripheral has a target channel adapter (TCA).[3] The Infiniband serial connection signaling rate is 2.5 Gbit/s in single data rate (SDR) technology, 5.0 Gbit/s in double data rate (DDR) technology or 10 Gbit/s in quad data rate (QDR), in each direction per connection. Moreover, the links can be aggregated in units of 4 or 12, designated as 4X and 12X. However, Infiniband uses 8B/10B encoding, which implies four fifths of the traffic is useful, therefore DDR 4X link carries 20 Gbit/s raw, or 16 Gbit/s of useful data. Table-1 summarizes the different Infiniband technologies with their associated theoretical performance numbers.

Table 1: Performance numbers of different Infiniband technologies

IB technology	SD IB Data Rate	DD IB Data Rate	QDR IB Data Rate
1x	2Gbps	4Gbps	8Gbps
4x	8Gbps	16Gbps	32Gbps
12x	24Gbps	48Gbps	96Gbps

Infiniband uses a hardware-offload protocol stack [3]. Extra memory copies that are sent from the application to an adapter can be avoided by the zero copy mechanism that optimizes the message transfer time. Moreover, Infiniband allows moving data from local memory to remote memory using RDMA (Remote Direct Memory Access), which allows the zero copy mechanism without involving the receiver host processor [2]. The number of user-kernel context switching and memory copies can be reduced by the direct access to the Infiniband HCA. Obviously, enabling communication between devices and hosts, without the traditional system resource overhead associated with network protocols, off-loads data movement from the server CPUs to the Infiniband HCA. Through virtual lanes (VLs), Infiniband offers traffic management, creating multiple virtual links within a single physical link that allows a pair of linked devices to isolate communication interference from other connected devices.

C. Intel Westmere Specifications

Westmere is the code name for the latest in the series of multi-core processors by Intel. This is Intel's true hexa-core processor with L2 cache sharing and utilizing the revolutionary Quick Path Interconnect (QPI) architecture [7] that provides two separate lanes for the communication between the CPU and the chipset. The QPI technology allows the CPU to transmit and receive I/O data in parallel, as opposed to the traditional architecture using a single external bus where the external bus is used for both input and output operations reads and writes cannot be done at the same time. The latest version of the QPI works with a clock rate of 3.2 GHz, transferring two data per clock cycle (Double Data Rate), making the bus to work as if it was using a 6.4 GHz clock rate.

Further, Intel Westmere generation is equipped with Turbo Boost Technology [7] that automatically allows processor cores to run faster than the base operating frequency if it's operating below power, current, and temperature specification limits. This frequency change is dependent on the number of active cores, estimated current consumption, estimated power consumption and processor temperature. When the processor is operating below these limits and the user's workload demands additional performance, the processor frequency will dynamically increase by 133 MHz on short and regular intervals until the upper limit is met or the maximum possible upside for the number of active cores is reached.

D. MVAPICH MPI Implementation

The Message Passing Interface (MPI) is the dominant programming model for parallel scientific applications. Given the role of the MPI library as the communication substrate for application communication, the library must ensure to provide scalability both in performance and in resource usage. In our experiments, we used MVAPICH, one of the most commonly used MPI implementations in the HPC industry. MVAPICH [12] implementation is mainly known for its support for Infiniband interconnect technologies as well as having high performance scalability support for clusters running thousands of cores. As for the Intel MPI, MVAPICH also supports various runtime environments such as SLURM and PBS.

III. PERFORMANCE EVALUATION AND RESULTS

To perform benchmark evaluation, a DELL cluster of PowerEdge M610 Blade Servers was used. The cluster consisted of 32 nodes with dual sockets and Intel hexa-Core x5670 (Westmere) 2.93GHz processors. The operating system running on the nodes was RedHat Enterprise Linux Server 5.3 with the 2.6.18-128.el5 kernel. Each node was equipped with an Infiniband Host Channel Adapter (HCA) supporting 4x Quad Data Rate (QDR) connections with the speed of 32Gbps. Each node also had 24 GB (6 x 4GB) DDR3 1333Mhz of memory, thus the total amount of memory the system had was around 786 GB.

The physical layout of the cluster consisted of two chassis, and each chassis hosts up to 16 blade nodes. From each node we had a 4x-QDR Infiniband connection going to a central 32-port Qlogic Infiniband switch. Figure 1 shows the Infiniband interconnection design as described. It is important to mention that this design is considered non-blocking as each node guarantees to have the full 4x QDR 32Gbps interconnect speed. This fast interconnect would drive the cluster to a higher utilization, which in theory, may affect the diskless concept.

Our Infiniband interconnect topology uses three switches: A top-level switch and other two leaf switches. Under this configuration, IPC communication among nodes of 12 sub-clusters is localized to one leaf switch, but for the cluster of 16 nodes, the top-level switch is involved to support more nodes.

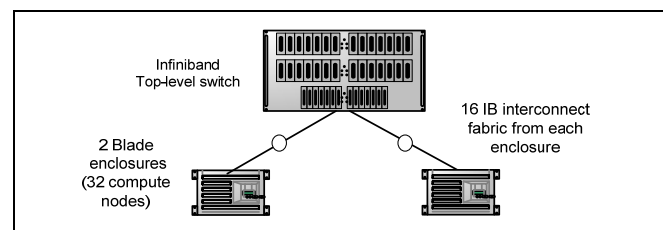


Figure 1: The DDR Infiniband interconnect for a 32 nodes cluster

In order to evaluate the performance of the two compilers, the benchmarks were run on the cluster nodes starting with one thread and scaling up to 12 threads for the OpenMP tests, and ranging from one node and up to 12 nodes for the MPI experiments.

In our experiments, we used two versions of matrix multiplication algorithms [13, 14] to benchmark the two

compilers. Beside it is computationally intensive with $O(n^3)$ iterations, we chose the matrix multiplication since it is a fundamental operation in many numerical linear algebra applications. Its efficient implementation on parallel computers is an issue of prime importance when providing such systems with scientific software libraries.

```

1.  #include <omp.h>
2.  #include <stdio.h>
3.  #include <stdlib.h>
4.  #define NRA 4000          /* # rows in matrix A */
5.  #define NCA 4000        /* # columns in matrix A */
6.  #define NCB 4000        /* # columns in matrix B */
7.  int main (int argc, char *argv[])
8.  {
9.      int      tid, nthreads, i, j, k, chunk;
10.     double   a[NRA][NCA], /* matrix A to be multiplied */
              b[NCA][NCB], /* matrix B to be multiplied */
              c[NRA][NCB]; /* result matrix C */
11.     chunk = 10;          /* set loop iteration chunk size */
12.     /*** Spawn a parallel region explicitly scoping all variables ***/
13.     #pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
14.     {
15.         tid = omp_get_thread_num();
16.         if (tid == 0)
17.         {
18.             nthreads = omp_get_num_threads();
19.             printf("Starting matrix multiple example with %d threads\n",nthreads);
20.             printf("Initializing matrices...\n");
21.         }
22.         /*** Initialize matrices ***/
23.         #pragma omp for schedule (static, chunk)
24.         for (i=0; i<NRA; i++)
25.         for (j=0; j<NCA; j++)
26.             a[i][j]= i+j;
27.         #pragma omp for schedule (static, chunk)
28.         for (i=0; i<NCA; i++)
29.         for (j=0; j<NCB; j++)
30.             b[i][j]= i*j;
31.         #pragma omp for schedule (static, chunk)
32.         for (i=0; i<NRA; i++)
33.         for (j=0; j<NCB; j++)
34.             c[i][j]= 0;
35.         /*** Do matrix multiply sharing iterations on outer loop ***/
36.         /*** Display who does which iterations ***/
37.         printf("Thread %d starting matrix multiply...\n",tid);
38.         #pragma omp for schedule (static, chunk)
39.         for (i=0; i<NRA; i++)
40.         {
41.             printf("Thread=%d did row=%d\n",tid,i);
42.             for(j=0; j<NCB; j++)
43.             for (k=0; k<NCA; k++)
44.                 c[i][j] += a[i][k] * b[k][j];
45.         } /*** End of parallel region ***/
46.         /*** Print results ***/
47.         printf("Result Matrix:\n");
48.         for (i=0; i<NRA; i++)
49.         {
50.             for (j=0; j<NCB; j++)
51.                 printf("%6.2f  ", c[i][j]);
52.             printf("\n");
53.         }
54.         printf ("Done.\n"); }

```

Figure 2: Matrix multiplication in C with OpenMP directives

Figure 2 shows the OpenMP C code for matrix multiplication. The routine `omp_get_num_threads` in line 15 is responsible for returning the number of threads that are currently in the team executing the parallel region from which it is called, while the `omp for (static, chunk)` schedule directive divides the iterations in the loop into pieces of size “chunk” and then statically assigns them to threads.

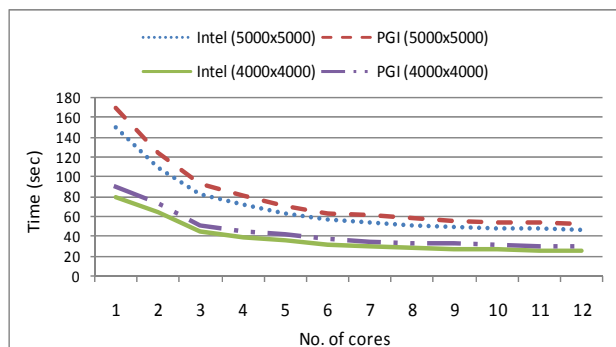


Figure 3: Intel vs. PGI using OpenMP directives in matrix multiplication

The OpenMP code was compiled using `-openmp` and `-mp` options for Intel and PGI compilers, respectively, while all other advance options were ignored to achieve a fair comparison. Figure 3 shows the performance benchmark of both Intel and PGI for multiplying 4000x4000 and 5000x5000 size matrices. Initially, all runs were significantly improved when adding more cores, while their improvement slowed down when reaching 6 cores. It was also observed that when the size of the matrices were increased from 4000 to 5000, the Intel-compiled code run time was increased by 79% in average, while PGI-compiled code was increased by 85%. In this OpenMP set of tests, the Intel compiler superseded PGI in all iterations.

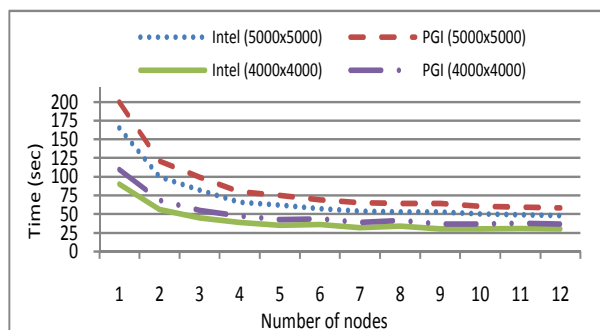


Figure 4: Intel vs. PGI using MPI in matrix multiplication

Figure 4 shows performance of the Intel and PGI compiled code using MPI routines. In this test, the C code (too long to be included in the paper, but can be found in [15]) was compiled using MVAPICH with Intel and PGI

parallel `mpicc` compilers. Similarly, all advance options were ignored. It is noticeable that the Intel compiled MPI-run on a single node/core took around 165 seconds, whereas it took only 150 seconds when running OpenMP on a single core. This is due to the fact that the MPI-based matrix multiplication C code has more routines and functions to call, making the code more complex, and thus more time to run. Another observation is the slight increase in the run time when multiplying the 4000x4000 size matrices on 11 and 12 cores. This increase is related to the additional communication overhead with respect to the computation time. This communication is lessened in the 5000x5000 multiplication as the computation time gets larger with respect to the communication overhead. Similar to the OpenMP test, the Intel compiler outperformed PGI in both 4000 and 5000 iterations.

To magnify the effect of MPI communication overhead with respect to computation time, we extended the MPI matrix multiplication benchmark runs to 32 nodes. Figure 5 shows the effect of this communication overhead as the number of nodes increases.

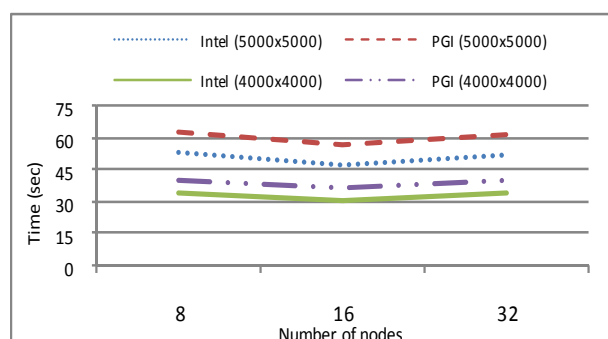


Figure 5: MPI scalability in 5000x5000 and 4000x4000 cells matrix multiplication using up to 32 nodes

Our next experiment was to evaluate the performance of the two compilers using a hybrid MPI/OpenMP environment to exploit the strength of both models. Specifically, the main driver for the hybrid parallel paradigm is to combine process level coarse-grain parallelism using MPI together with fine-grain parallelism on a loop level using OpenMP. Obviously, finding the right combination of cores/nodes to gain the best performance depends on the nature of the application and can only be found by performing empirical runs. As previously observed in figure 3, the OpenMP runs started to saturate when exceeding 8 cores. Based on this statement and since the matrix multiplication algorithm is compute-bound, we varied the number of cores in our next experiment between 4 and 12 cores, while varying the number of nodes between 1 and 32 to obtain the right cores-to-nodes (i.e. OpenMP-to-MPI) ratio that gives the best performance numbers. Figure 6 shows the evaluation performance of running this hybrid code using 5000x5000 cells matrices. Noticeably, the best

ratio was found when running on 8 nodes with 4 cores in each node (39 seconds), while the worst combination was resulted when running on 32 nodes and 12 cores in each node (180 seconds) (a total of 384 processes were spawned).

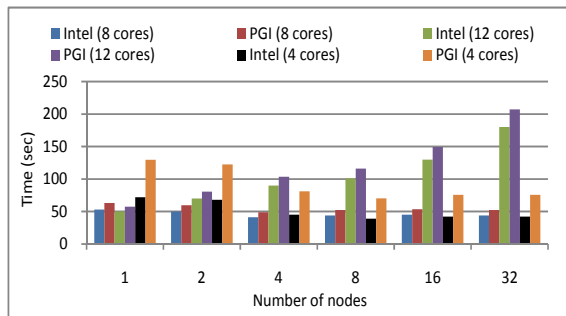


Figure 6: Intel vs. PGI using MPI in matrix multiplication

IV. CONCLUSION

Intel and Portland Group have been designing their parallel compilers to leverage the rich set of performance enabling features in modern CPUs and parallel systems. This is achieved by tightly integrating OpenMP directives and advanced MPI optimizations to generate efficient multithreaded code for exploiting parallelism at various levels. In this paper, we evaluated the performance of two commonly used parallel compilers, Intel and Portland's PGI, using a state-of-the-art Intel Westmere-based HPC cluster. The performance evaluation was based on two sets of experiments, once evaluating the compilers' performance using an MPI-based code, and another using OpenMP. Our results show that, for scientific applications that are matrices-dependant, the MPI and OpenMP features of the Intel compiler supersede PGI when using the defined HPC cluster.

ACKNOWLEDGMENT

We thank KFUPM for their support.

REFERENCES

- [1] R. AlShaikh, M. Ghuson, M. Baddourah, "Performance Evaluation of Myrinet and Cisco Infiniband Using Intel MPI Middleware", the 9th LCI International Conference on High Performance Computing, NCSA, University of Illinois, USA, May 2008.
- [2] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda, "Host-Assisted Zero-Copy Remote Memory Access Communication on InfiniBand", Int'l Parallel and Distributed Processing Symposium (IPDPS 04), April, 2004.
- [3] C. Bell, D. Bonachea, Y. Cote and et al. "An Evaluation of Current High-Performance Networks", Int'l Parallel and Distributed Processing Symposium (IPDPS'03), April 2003.
- [4] J. Liu, B. Chandrasekaran, J. Wu and et al. "Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics", Supercomputing, ACM/IEEE, pages 58- 58, Nov. 2003.
- [5] Myrinet, Myricom. Available at: <http://www.myri.com>
- [6] R. Fatoohi, K. Kardys, S. Koshy and et al. "Performance evaluation of high-speed interconnects using dense communication patterns", Parallel Computing Volume 32, Issue 11-12, pages 794-807, 2006.
- [7] Intel Inc. Available at: <http://www.intel.com>
- [8] Portland PGI. Available at: <http://www.pgroup.com/>
- [9] The top500 supercomputers. Available at: <http://www.top500.org>
- [10] MVAPICH: MPI over InfiniBand and iWARP. Available at: <http://mvapich.cse.ohio-state.edu>
- [11] T. Typou, V. Stefanidis, P.D. Michailidis and K.G. " Margaritis, Implementing Matrix Multiplication on an MPI Cluster of Workstations", in Proceedings of the 1st In't Conference "From Scientific Computing to Computational Engineering" (IC-SCCE'2004), Athens, Greece, vol. II, pp. 631-639, 2004
- [12] B. Madani, R. Al-Shaikh, "Performance Modeling and MPI Evaluation Using Westmere-based Infiniband HPC Cluster", 4th European Modelling Symposium on Mathematical Modelling and Computer Simulation, Pisa, Italy, 2010
- [13] Lawrence Livermore National Laboratory – OpenMP tutorial. Available at: <https://computing.llnl.gov/tutorials/openMP/>
- [14] Simple matrix multiplication on MPI. Available at: <http://sushpa.wordpress.com/2008/05/20/simple-matrix-multiplication-on-mpi/>

Predictive and Distributed Routing Balancing for HPC Clusters

Carlos Núñez Castillo, Diego Lugones, Daniel Franco, and Emilio Luque

Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona, Spain

Abstract—Current parallel applications in parallel computing systems require an interconnection network to provide low and bounded communication delays. Communication characteristics such as traffic pattern and communication load change over time and, eventually, they may exceed network capacity causing congestion and performance degradation. Congestion control based on adaptive routing should be applied in order to adapt quickly to changing traffic conditions. Studies of parallel applications show repetitive behavior and that they can be characterized by a set of representative phases. This work presents a Predictive and Distributed Routing Balancing technique (PR-DRB) to control network congestion based on adaptive traffic distribution. PR-DRB uses speculative routing based on application repetitiveness. PR-DRB monitors messages latencies on routers and logs solutions to congestion, to quickly respond in future similar situations. Experimental results show that the predictive approach could be used to improve performance.

Keywords: Interconnection networks, predictive routing, parallel applications, application aware routing.

1. Introduction

In the early days of High Performance Computing (HPC) systems, interconnection network high latency and low bandwidth bottleneck significantly affected applications execution. Advances in technologies such as InfiniBand (IBA) [1] allowed higher transmission rate and lower latency. HPC communications are characterized by bursty traffic [2]. Bursty traffic can produce Hot-Spot situations, where some network resources are congested while others remains idle. If congestion is not efficiently controlled, message latency is increased and performance is degraded. Communication patterns in HPC applications are repetitive [3]. This repetitiveness could be useful to the routing module to solve future network congestions. We propose a Predictive and Distributed Routing Balancing algorithm (PR-DRB) after considering routing algorithm limitations and requirements together with applications repetitiveness. Our main goal is to reduce latency under repetitive communication patterns. PR-DRB is based on DRB [4], but enhanced with a predictive routing module. DRB adapt itself to congestion by opening alternative paths. This stabilization process is costly in time.

The main contributions of this work is the capability to learn from a parallel application communication pattern, solve congestion and then use this solution when similar congestion is detected again. Repetitive communication

patterns alternated with computation is a typical HPC application feature[2], and it represents an application phase [3]. Applications alternate between phases, which causes specific traffic patterns (e.g. a set of source/destinations pairs) to reappear. PR-DRB strategy is shown in Fig. 1. During application first phase, PR-DRB has high latency values (1) because it is searching alternative paths. At the end of phase 1 (2), latency is stable and the best solutions found are saved at the source node. Best solutions are identified when latency curve starts decreasing. Later phases do not reach its highest historical latency value. Here, PR-DRB has identified similar communication patterns again(3) and best paths saved are used(4). PR-DRB approach is to maintain stable latency values during the whole application execution. The rest of this paper is organized as follows. In Sec. 2 congestion control, parallel application repetitiveness and their relation to this work are given. In Sec. 3 the PR-DRB methodology is described. Sec. 4 shows the performance evaluation. Conclusion are explained in Sec. 5.

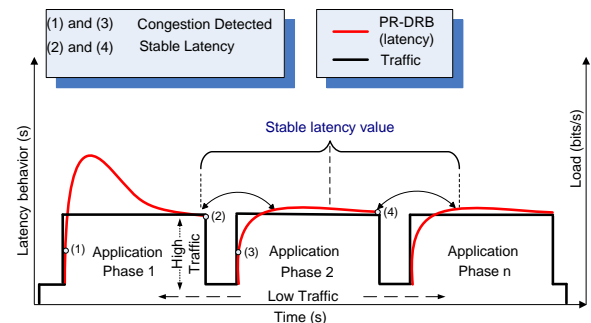


Fig. 1: PR-DRB-Process

2. Background and Justification

2.1 Congestion Control

Congestion control is based on monitoring, detection and further control. To evaluate congestion alternatives such as point to point latency [5], buffer occupation level [6] or backpressure [7] could be used. Message Throttling [8] stops (or reduces) packet injection but latency is increased because packets must remain at source nodes longer. Adaptive routing algorithms [4], [9] work by sending messages from source to destination through alternative paths. In adaptive routing congested area is avoided and message injection is upheld. Monitoring overhead, path changing and the need to guarantee both: deadlock freedom [10] and 'in-order' packet delivery are some of its disadvantages.

2.2 Parallel Application Repetitiveness

Studies of parallel applications in HPC reveal they have repetitive behavior, based on computing and communications phases [3]. Programs have a very strong periodic behavior [11]. On Fig. 2 the repetitive behavior of the NAMD application is shown. Repetitive behavior is represented by fundamental phases of the entire application (e.g. a set of source/destination pairs). For example, the NAS CG benchmark has 4 representative phases and they consume 99.10% of execution time. Each phase here is repeated 2600 times on average during execution. The SMG2000 also has 4 representative phases, and they consume 99.99% of total execution time. Here, phases 1 to 4 are repeated 1, 1, 1185 and 15 times, respectively. Here only phase 3 is relevant to communications. SWEEP3D has approximately 80 different phases, but only 5 phases are representative by consuming 96.17% of total time. Representative phases were extracted with the PAS2P tool [3].

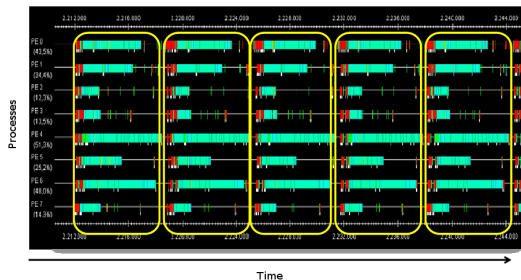


Fig. 2: Repetitiveness in Parallel Applications

2.3 Justification

Based on previous examples of communication patterns repetitiveness, we can say that High Speed Interconnection Networks (HSIN) routing performance depends mostly on the communication pattern used and the mapping of nodes to processors. To improve communication performance, hence applications currently running in the network, a technique capable to dynamically combine adaptive algorithms and communication patterns is needed, so that routing and congestion control can perform as fast as possible and minimize overhead.

3. Predictive-Distributed Routing Balancing

We propose a routing algorithm, PR-DRB, based on the study of communication latencies and repetitive application patterns in HPC applications. PR-DRB internals is covered here in more details.

3.1 PR-DRB Working Scheme

PR-DRB seeks better response time by using cached communication and alternative paths. The proposed model

performs four basic tasks: Monitoring, Notification, Path Configuration and Path selection procedures. Monitoring includes the tasks of latency values accumulation and contending flows identification, performed at intermediate routers. Notification is initiated at destination endnodes. Here, and Acknowledge (ACK) message with path information is created and sent back to the source. The third task involves the configuration of new alternative paths (Metapath Configuration) according to latency values, also performed at source nodes. If there are saved solutions for a congestion situation, the paths are taken from the saved solution database. Otherwise, new alternative paths are created. Later, the fourth task is accomplished when new messages are injected into the network. Here, selection procedures distributes messages among the paths configured in previous task. Fig. 3 shows PR-DRB functionality and its tasks. When a source node wants to send some data, depicted in Source Endnode, a message is built and injected into the network. Then, as seen in Message Routing, the multi-header message is forwarded through intermediate routers. As shown in the Monitoring box in Fig. 3, the delay suffered in switch buffers (queuing latency) is logged into the message. If queuing latency values exceeds a threshold while still at intermediate routers, contending flows patterns are also logged by PR-DRB. Once the message reaches destination, as seen in Destination Endnode, Notification takes place. The Notification box depicts the task involved in this procedure. Here, latency as well as contending communication patterns found are sent back to the source in an ACK. Not all contending flows are notified, but only those which contributes most to congestion. At source nodes latency value and contending flows are analyzed as shown in the Fig. 3, at the Metapath Configuration box. This module configures alternative paths to be used accordingly to latency value. If latency denotes congestion, then new alternative paths are needed. PR-DRB then looks for an already analyzed congestion situation. If this is the case, the set of optimal alternative paths used previously is obtained from the database. If no solutions are found, then alternative paths opening procedures are initiated. If latency values denotes congestion stabilization, then alternative paths closing procedures are invoked. Here, information about contending flows during congestion situations is also updated. Later, when a message is ready to be injected into the network PR-DRB performs the Path Selection. Here, PR-DRB selects which paths are going to be used from those configured in the Metapath Configuration step. Paths having lower latency values are more frequently used, and they receive proportionally a greater number of messages. Given a source node with N alternative paths, let's be $L_c(i : 1..N)$ the latency recorded by path C_i . The alternative path C_x will be selected in the following injection

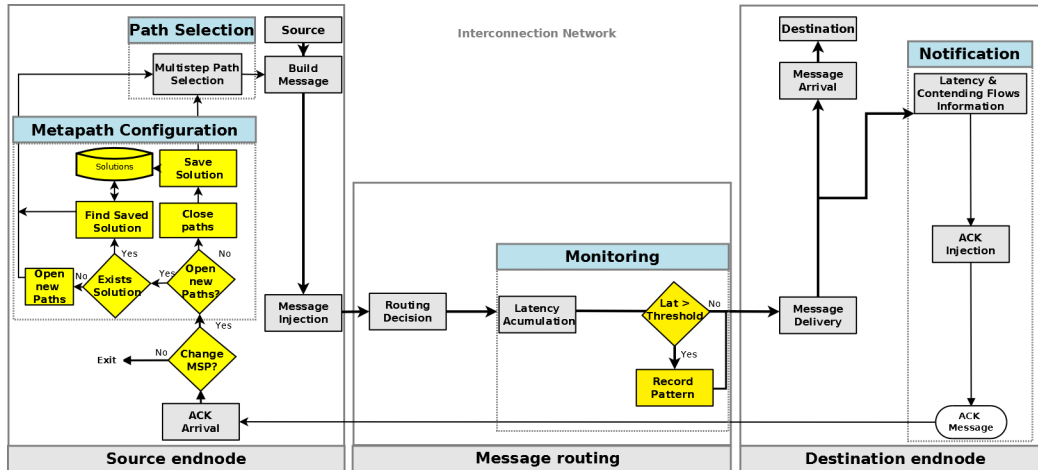


Fig. 3: PR-DRB-Process

according to the probability:

$$p(Cx) = \frac{(1/L_Cx)}{\sum_{i=1}^N 1/L_Ci} \quad (1)$$

The process of detecting already analyzed situations is based on contending flows similarity, which is based on approximation matching. As shown in Fig. 3, a message is forwarded without any overhead when the output port is free. Otherwise, packet is queued and latency is simultaneously accumulated until the message is ready to be forwarded again. PR-DRB is based on the DRB algorithm, and already proposed congestion control for Infiniband [10] could be used. As IBA already has functionalities required by PR-DRB (e.g. monitoring functions at IBA switches, the CCA has procedures for congestion notification and path opening), PR-DRB integration is feasible. The following section presents the performance evaluation of PR-DRB policy. The evaluation is designed to compare PR-DRB behavior against DRB [4], which has been already compared against other traditional algorithms, under different interconnection network scenarios.

4. PR-DRB Evaluation

Latency is evaluated in order to assess PR-DRB. Latency is the time elapsed since a packet is created until it reaches its destination, in seconds. Evaluation was conducted for fat tree topologies with 32 and 64 nodes.

4.1 Modeling Environment

PR-DRB operations together with network components were modeled [12] using the standard simulation and modeling tool OPNET [13]. OPNET provides a Discrete Event Simulator engine. This allows defining network components behavior, and it supports detailed specification of protocols. We have assumed virtual Cut-through flow control [14]. Link Bandwidth was set to 2Gbps, packet size was set to 1024 bits and the size of routers buffers was 2MB.

4.2 Analysis with Permutation Traffic

PR-DRB is evaluated under the fat tree topology with 32 and 64 nodes. Communication patterns used are: "Matrix Transpose" and "Perfect Shuffle". Fig. 4 and 5 shows the performance under Matrix transpose pattern, for 32 nodes and traffic load from 400 to 600 mbps/node respectively. PR-DRB latency reduction achieved is 24% under both scenarios. Proper communication balancing procedures and packets sent through optimal alternative paths from the beginning, keep congestion at minimum. Under 600 mbps/node injection, PR-DRB uses progressively the maximum number of alternative paths to deliver messages. For repetitive traffic patterns, maximum path expansion is done directly. By avoiding intermediate path expansion, unnecessary ACK messages are avoided and overhead is minimized. With at most 4 alternative paths, PR-DRB performs a remarkable lower latency than DRB. Fig. 6 shows results with 64 communicating nodes. Latency reduction under the perfect shuffle pattern is 32%. Fig. 7 shows the Matrix Transpose pattern results. Higher load is injected and latency remains bounded. Latency is reduced here around 40% compared to DRB. Recall that PR-DRB will behave similarly to DRB only under the first phase of the application. In this stage PR-DRB is learning from the path opening procedures. In later phases of parallel applications, like those shown here, PR-DRB will apply directly the best solutions saved. From time 1.015 latency values of both algorithms tend to become stable and converge.

5. Conclusion

We proposed the Predictive and Distributed Routing Balancing, PR-DRB, which uses alternative paths to reduce latency by considering traffic dynamic behavior constraints. Applications that run on an HSIN possess repetitive behavior, and PR-DRB is capable to learn from it and save information for later use. PR-DRB has been developed to

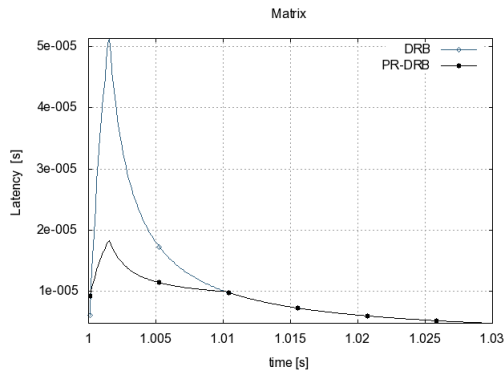


Fig. 4: Permutation patterns - Fat tree 32 nodes - 400 mbps

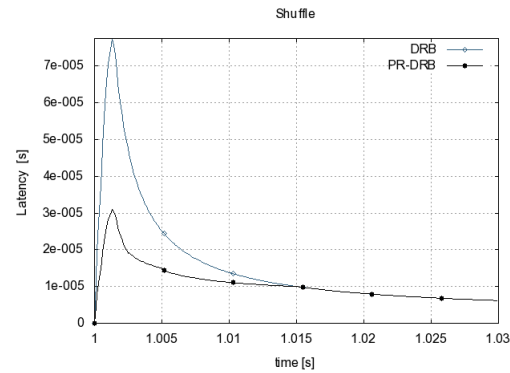


Fig. 6: Permutation patterns - Fat tree 64 nodes - 400 mbps

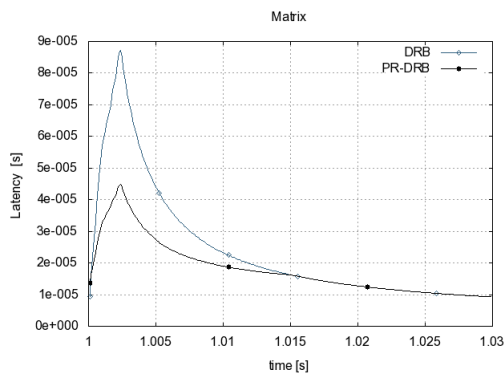


Fig. 5: Permutation patterns - Fat tree 32 nodes - 600 mbps

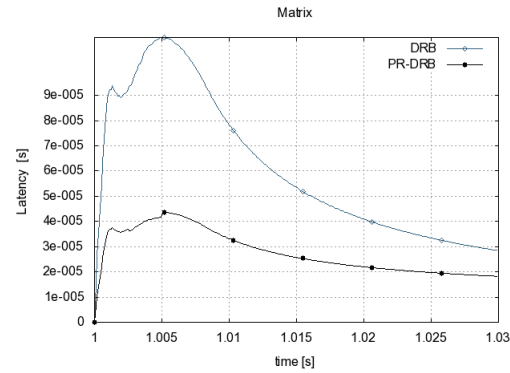


Fig. 7: Permutation patterns - Fat tree 64 nodes - 600 mbps

fulfill HSIN objectives such as all-to-all connection, and low and uniform latency under any traffic load. The proposed method is in line with commercial interconnects (as InfiniBand [1]). Our policy allows heavier communication load in the network, or in cost-bounded data centers it allows using less network components, because they are efficiently handled. The evaluation performed to validate PR-DRB has revealed improvements in latency. We have shown that PR-DRB is a fast and robust method. PR-DRB is useful for bursty communication patterns, which are commonly created by parallel applications and can produce the worst hot-spot situations. As future work, we plan to predict congestion before it appears by analyzing latency trend.

ACKNOWLEDGMENT

This work has been supported by the MEC-MICINN Spain under contract TIN2007-64974. Also, we thank OPNET Technologies Inc. for the OPNET Modeler licenses.

References

- [1] Infiniband, "Iba," <http://www.infinibandta.org/>, 2011.
- [2] G. Rodriguez *et al.*, "Exploring pattern-aware routing in generalized fat tree networks," in *ICS '09: Procs of the 23rd int. conf. on Supercomp.* USA: ACM, 2009, pp. 276–285.
- [3] A. Wong *et al.*, "Parallel application signature," *CLUSTER '09. IEEE Int. Conf. on*, vol. 1, pp. 1–4, 2009.

- [4] D. Franco *et al.*, "A new method to make communication latency uniform: distributed routing balancing," in *ICS '99: Procs of the 13th int. conf. on Superc.* USA: ACM, 1999, pp. 210–219.
- [5] D. Lugones *et al.*, "Dynamic and distributed multipath routing policy for high-speed cluster networks," in *CCGRID '09: Procs of the 2009 9th IEEE/ACM Int. Symp. on Cluster Comp. and the Grid*, USA, 2009, pp. 396–403.
- [6] P. Garcia *et al.*, "Recn-dd: A memory-efficient congestion management technique for advanced switching," *Parallel Proc., Int. Conf. on*, vol. 0, pp. 23–32, 2006.
- [7] E. Baydal *et al.*, "A family of mechanisms for congestion control in wormhole networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 9, pp. 772–784, 2005.
- [8] S. Yan *et al.*, "An enhanced congestion control mechanism in infiniband networks for high performance computing systems," *Adv. Inf. Networks and App., Int. Conf. on*, vol. 1, pp. 845–850, 2006.
- [9] A. Singh *et al.*, "Globally adaptive load-balanced routing on tori," *IEEE Comput. Archit. Lett.*, vol. 3, no. 1, p. 2, 2004.
- [10] D. Lugones *et al.*, "Dynamic routing balancing on infiniband networks," in *Journal of Comp. Science & Tech. (JCS&T)*, ser. Cluster Computing '08, 2008, pp. 104–110.
- [11] T. Sherwood *et al.*, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *PACT '01: Procs of the 2001 Int. Conf. on Par. Arch. and Compil. Tech.* USA: IEEE Comp. Soc., 2001, pp. 3–14.
- [12] D. Lugones *et al.*, "Modeling adaptive routing protocols in high speed interconnection networks," *OPNETWORK 2008 Conf.*, 2008.
- [13] T. OPNET, "Opnet modeler accelerating network r&d," <http://www.opnet.com>, June 2008, opNET.
- [14] J. Duato *et al.*, *Interconnection Networks: An Engineering Approach*. USA: M. Kaufmann Pub. Inc., 2002.

A parallel algorithm for the verification of Covering Arrays

Himer Avila-George¹, Jose Torres-Jimenez², Vicente Hernández¹ and Nelson Rangel-Valdez²

¹Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, Valencia, Spain

²Laboratorio de Tecnologías de Información, CINVESTAV-Tamaulipas, Ciudad Victoria, Tamaulipas, Mexico

Abstract—Covering Arrays (CAs) are combinatorial objects that, with a small number of cases, cover a certain level of interaction of a set of parameters. CAs have found application in a variety of fields where interactions among factors need to be identified; some of these fields are biology, agriculture, medicine, and software and hardware testing. In particular, a covering array is an $N \times k$ matrix over an alphabet v s.t. each $N \times k$ subset contains at least one time each combination from $\{0, 1, \dots, v - 1\}^t$, given a positive integer value t . The process of ensuring that a CA contains each of the v^t combinations is called verification of CA. When CAs have many variables or their strength is greater than 3, its verification is computationally very expensive. In this paper we present an algorithm for CA verification and its implementation details in sequential and parallel computing.

Keywords: Covering Arrays, Parallel Computing Algorithms, Software Testing

1. Introduction

The experimental design is a key piece in the software development given that it allows to identify fails in the software before it begins to operate. A good strategy to test a software involves the generation of the whole set of cases that participate in its operation. However, testing all the possible cases of a program requires a great amount of time, which can be inadmissible even for small programs [1].

An alternative strategy to test a software is the use of Covering Arrays (CAs). A Covering array (CA) is a combinatorial object that, with a small number of cases, covers a certain level of interaction of a set of parameters. The meaning of level of interaction relates any subset of t parameters of a matrix to the set of the v^t different combinations derived from v different values. The confidence level of the testing using combinatorial objects as CA increases with the interaction level involved [2].

Covering arrays (CAs) have been object of study and application in different research areas. Cawse [3] used CAs in the material design, Hedayat *et al.* [4] used them in medicine and agriculture; in biology and industrial processes have also been used by Shasha *et al.* [5] and Pahdke [6]. CAs have been used in hardware testing [7] but significantly the area with the major application of these objects is in software

testing [8], [9]. Next, Definition 1.1 is a formal definition of CA.

Definition 1.1 (Covering Array): Let N, t, k, v be positive integers with $t \leq k$. A covering array, denoted by $CA(N; t, k, v)$, of alphabet v , strength t , is an array \mathcal{M} of size $N \times k$, where each element $m_{i,j}$ takes values from the set $S = \{0, 1, 2, \dots, v - 1\}$ and each subset of \mathcal{M} of size $N \times t$ contains all the possible combinations derived from $\{0, 1, \dots, v - 1\}^t$ symbols. In the rest of this document a subset of size $N \times t$ will be known as a t -tuple and the initial t -tuple will be $\{1, 2, \dots, t\}$.

To illustrate the CA approach applied to the design of software testing, consider the Web-based system example shown in Table 1, the example involves four parameters each with three possible values. A full experimental design ($t = 4$) should cover $3^4 = 81$ possibilities, however, if the interaction is relaxed to $t = 2$ (pair-wise), then the number of possible combinations is reduced to 9 test cases.

Table 1: Parameters of Web-based system example.

	Browser	OS	DBMS	Connections
0	Firefox	Windows 7	MySQL	ISDN
1	Chromium	Ubuntu 10.10	PostgreSQL	ADSL
2	Netscape	Red Hat 5	MaxDB	Cable

Figure 1 shows the CA corresponding to $CA(9; 2, 4, 3)$; given that its strength and alphabet are $t = 2$ and $v = 3$, respectively, the combinations that must appear at least once in each subset of size $N \times 2$ are $\{0, 0\}$, $\{0, 1\}$, $\{0, 2\}$, $\{1, 0\}$, $\{1, 1\}$, $\{1, 2\}$, $\{2, 0\}$, $\{2, 1\}$, $\{2, 2\}$.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 2 & 2 & 2 \\ 1 & 0 & 1 & 2 \\ 1 & 1 & 2 & 0 \\ 1 & 2 & 0 & 1 \\ 2 & 0 & 2 & 1 \\ 2 & 1 & 0 & 2 \\ 2 & 2 & 1 & 0 \end{pmatrix}$$

Fig. 1: A combinatorial design, $CA(9; 2, 4, 3)$.

Finally, to make the mapping between the CA and the Web-based system, every possible value of each parameter in Table 1 is labeled by the row number. Table 2 shows the corresponding pair-wise test suite; each of its nine experiments is analogous to one row of the CA shown in Figure 1.

Table 2: Test-suite covering all 2-way interactions, $CA(9; 2, 4, 3)$.

Experiments				
1	Firefox	Windows 7	MySQL	ISDN
2	Firefox	Ubuntu 10.10	PostgreSQL	ADSL
3	Firefox	Red Hat 5	MaxDB	Cable
4	Chromium	Windows 7	PostgreSQL	Cable
5	Chromium	Ubuntu 10.10	MaxDB	ISDN
6	Chromium	Red Hat 5	MySQL	ADSL
7	Netscape	Windows 7	MaxDB	ADSL
8	Netscape	Ubuntu 10.10	MySQL	Cable
9	Netscape	Red Hat 5	PostgreSQL	ISDN

When a CA contains the minimum possible number of rows, it is optimal and its size is called the *Covering Array Number* (CAN). The CAN is defined according to

$$CAN(t, k, v) = \min\{N : \exists CA(N; t, k, v)\}.$$

The trivial mathematical *lower bound* for a covering array is $v^t \leq CAN(t, k, v)$, however, this number is rarely achieved. Therefore determining achievable lower bounds is one of the main research lines for CA s.

The construction of $CAN(2, k, 2)$ can be efficiently done according with [10]; the same is possible for $CA(2, k, v)$ when the cardinality of the alphabet is $v = p^n$, where p is a prime number and n a positive integer value [11]. However, in the general case determining the *covering array number* is known to be a hard combinatorial problem [12], [13]. For the values of t and v that no efficient algorithm is known, we use approximated algorithms to construct them. Some of these approximated strategies must verify that the matrix they are building is a CA . If the matrix is of size $N \times k$ and the interaction is t , there are $\binom{k}{t}$ different combinations which implies a cost of $O(N \times \binom{k}{t})$ for the verification (when the matrix has $N \geq v^t$ rows, otherwise it will never be a CA and its verification is pointless). For small values of t and v the verification of CA s is overcome through the use of sequential approaches; however, when we try to construct CA s of moderate values of t , v and k , the time spent by those approaches is impractical. Then, the necessity for parallel algorithms to construct and verify CA s appears.

In this paper we propose an algorithm to verify that a given matrix is a CA . The algorithm is implemented following the sequential and parallel strategies. The performance of each paradigm is experimentally compared with the purpose of showing their particular limitations. The remaining of the paper is organized as follows. Section 2 reviews related work dedicated to the construction and verification of CA s. Section 3 formally presents the problem of verifying a CA and describes the approaches proposed in this paper to verify CA s. Section 4 presents the methodology followed to experimentally compare the approaches; this section ends with some discussion derived from the results obtained from the experiment. Section 5 presents the conclusions derived from the research presented in this paper.

2. Relevant related work

Because of the importance of the construction of (near) optimal CA s, 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: (a) algebraic methods, (b) recursive methods, (c) greedy methods, and (d) meta-heuristics. In this section we describe the relevant related work to the construction of CA s.

Algebraic approaches construct CA s using predefined rules. Most algebraic approaches compute CA s directly by a mathematical function [14], [15]. The algorithms that use recursive constructions generate a number of small CA s and with them build CA s of greater size. These algorithms have been used in Augmented Annealing [16] and CTS [17] to construct CA s. The majority of commerce and open source test data generating tools use greedy algorithms for covering array construction (AETG [18], TCG [19], DDA [20], ACTS [12], [21], and IRPS [22]). Some meta-heuristics that have been used to solve the CA problem are: Simulated Annealing (SA) [23], [24], Tabu Search (TS) [25], [26] and Memetic Algorithms (MA) [27].

All the algorithms already presented that construct covering arrays must contain a certificate that the resulting matrix is indeed a CA . For general matrices, the certificate is given by the verification process of a CA ; this process requires to test that the sub-matrices derived from the $\binom{k}{t}$ different t -tuples contain all the combinations of symbols found in $\{0, 1, \dots, v-1\}^t$. Hence, the cost of verifying a CA is a time consuming task that is highly combinatorial and rapidly becomes impractical to be solved by sequential approaches (even for moderate values of t, k, v of the CA). It is there where lies the necessity of a parallel strategy that makes faster the verification process in larger CA s.

The next section presents sequential and parallel strategies proposed in this paper to verify a given matrix as a CA .

3. How to verify CA s

A matrix \mathcal{M} of size $N \times k$ is a $CA(N; t, k, v)$ iff every t -tuple contains the set of combination of symbols described by $\{0, 1, \dots, v-1\}^t$. We propose a strategy that uses two data structures called P and J , and two injections between the sets of t -tuples and combinations of symbols, and the set of integer numbers, to verify that \mathcal{M} is a CA .

Let $\mathcal{C} = \{c_1, c_2, \dots, c_{\binom{k}{t}}\}$ be the set of the different t -tuples. A t -tuple $c_i = \{c_{i,1}, c_{i,2}, \dots, c_{i,t}\}$ is formed by t numbers, each number $c_{i,1}$ denotes a column of matrix \mathcal{M} . The set \mathcal{C} can be managed using an injective function $f(c_i) : \mathcal{C} \rightarrow \mathcal{I}$ between \mathcal{C} and the integer numbers, this function is defined in Equation 1.

$$f(c_i) = \sum_{j=1}^t \binom{c_{i,j} - 1}{i + 1} \quad (1)$$

Now, let $\mathcal{W} = \{w_1, w_2, \dots, w_{v^t}\}$ be the set of the different combination of symbols, where $w_i \in \{0, 1, \dots, v-1\}^t$. The injective function $g(w_i) : \mathcal{W} \rightarrow \mathcal{I}$ is defined as done in Equation 2. The function $g(w_i)$ is equivalent to the transformation of a v -ary number to the decimal system.

$$g(w_i) = \sum_{j=1}^t w_{i,j} \cdot v^{t-j} \quad (2)$$

The use of the injections represents an efficient method to manipulate the information that will be stored in the data structures P and J used in the verification process of \mathcal{M} as a CA. The matrix P is of size $\binom{k}{t} \times v^t$ and it counts the number of times that each combination appears in \mathcal{M} in the different t -tuples. Each row of P represents a different t -tuple, while each column contains a different combination of symbols. The management of the cells $p_{i,j} \in P$ is done through the functions $f(c_i)$ and $g(w_j)$; while $f(c_i)$ retrieves the row related with the t -tuple c_i , the function $g(w_j)$ returns the column that corresponds to the combination of symbols w_j . The vector J is of size t and it helps in the enumeration of all the t -tuples $c_i \in \mathcal{C}$.

Table 3 shows an example of the use of the function $g(w_j)$ for the Covering Array $CA(9; 2, 4, 3)$ (shown in Figure 1). Column 1 shows the different combination of symbols. Column 2 contains the operation from which the equivalence is derived. Column 3 presents the integer number associated with that combination.

Table 3: Mapping of the set \mathcal{W} to the set of integers using the function $g(w_j)$ in $CA(9; 2, 4, 3)$ shown in Figure 1.

\mathcal{W}	$g(w_i)$	\mathcal{I}
$w_1 = \{0,0\}$	$0 \cdot 3^1 + 0 \cdot 3^0$	0
$w_2 = \{0,1\}$	$0 \cdot 3^1 + 1 \cdot 3^0$	1
$w_3 = \{0,2\}$	$0 \cdot 3^1 + 2 \cdot 3^0$	2
$w_4 = \{1,0\}$	$1 \cdot 3^1 + 0 \cdot 3^0$	3
$w_5 = \{1,1\}$	$1 \cdot 3^1 + 1 \cdot 3^0$	4
$w_6 = \{1,2\}$	$1 \cdot 3^1 + 2 \cdot 3^0$	5
$w_7 = \{2,0\}$	$2 \cdot 3^1 + 0 \cdot 3^0$	6
$w_8 = \{2,1\}$	$2 \cdot 3^1 + 1 \cdot 3^0$	7
$w_9 = \{2,2\}$	$2 \cdot 3^1 + 2 \cdot 3^0$	8

The matrix P is initialized to zero. The construction of matrix P is direct from the definitions of $f(c_i)$ and $g(w_j)$; it counts the number of times that a combination of symbols $w_j \in \mathcal{W}$ appears in each subset of columns corresponding to a t -tuple c_i , and increases the value of the cell $p_{f(c_i), g(w_j)} \in P$ in that number.

Table 4b shows the use of injective function $f(c_i)$. Table 4b presents the matrix P of $CA(9; 2, 4, 3)$. The different combination of symbols $w_j \in \mathcal{W}$ are in the first rows. The number appearing in each cell referenced by a pair (c_i, w_j) is the number of times that combination w_j appears in the set of columns c_i of the matrix $CA(9; 2, 4, 3)$.

In summary, to determine if a matrix \mathcal{M} is or not a CA the number of different combination of symbols per t -tuple

is counted using the matrix P . The matrix \mathcal{M} will be a CA iff the matrix P contains no zero in it.

Several approaches can be followed to implement this strategy to verify a CA. The traditional one is the sequential algorithm (one instruction at a time), other approach is parallel computing. These strategies use the data structures described in this section and are discussed in the following subsections.

3.1 Sequential Algorithm (SA) to verify CAs

The Sequential Algorithm (SA) takes as input a matrix \mathcal{M} and the parameters N, k, v, t that describe the CA that \mathcal{M} can be. Also, the algorithm requires the sets \mathcal{C} and \mathcal{W} and, without lost of generality, the values \mathcal{K}_l and \mathcal{K}_u that represent the first and last t -tuple to be verified (which for SA are $\mathcal{K}_l = 1, \mathcal{K}_u = \binom{k}{t}$). SA outputs the total number of missing combinations in the matrix \mathcal{M} to be a CA. The algorithm first counts for each different t -tuple c_i the times that a combination $w_j \in \mathcal{W}$ is found in the columns of \mathcal{M} corresponding to c_i . After that, SA calculates the missing combinations $w_j \in \mathcal{W}$ in c_i . Finally, the algorithm transforms c_i into c_{i+1} , i.e. it determines the next t -tuple to be evaluated.

The pseudo-code for SA is presented in Algorithm 1. For each different t -tuple (lines 5 to 21) the algorithm performs the following actions: counts the expected number of times a combination w_j appears in the set of columns indicated by J (lines 6 to 11, where the combination w_j is the one appearing in $\mathcal{M}_{n,J}$, i.e. in row n and t -tuple J); then, the counter *covered* is increased in the number of different combinations with a number of repetitions greater than zero (lines 10 and 11). After that, the algorithm calculates the number of missing combinations (line 12). The last step of each iteration of the algorithm is the calculation of the next t -tuple to be analyzed (lines 13 to 21). The algorithm ends when all the t -tuples have been analyzed (line 5).

3.2 Parallel Approach (PA) to verify CAs

The parallel strategy to verify CAs is simple. It involves a block distribution model of the set of t -tuples. The set \mathcal{C} is divided into n blocks, where n is the processors number; the size of block \mathcal{B} is equal to $\lceil \frac{|\mathcal{C}|}{n} \rceil$. The block distribution model maintains the simplicity in the code; this model allows the assignment of each block to a different core such that SA can be applied to verify the blocks.

To make the distribution of work, it is necessary to calculate the initial t -tuple f for each core according to its ID (denoted by *rank*), where $F = \text{rank} \cdot \mathcal{B}$. Therefore it is necessary a method to convert the scalar F to the equivalent t -tuple $c_i \in \mathcal{C}$. The sequential generation of each t -tuple c_i previous to c_F can be a time consuming task. There is where lies the main contribution of our parallel approach; its simplicity is combined with a clever strategy for computing the initial t -tuple of each block.

Table 4: Example of the matrix P resulting from $CA(9; 2, 4, 3)$ presented in Figure 1.

(a) Applying $f(c_i)$.			(b) Matrix P .								
index	c_i t-tuple	$f(c_i)$	$f(c_i)$	{0,0}	{0,1}	{0,2}	{1,0}	$g(w_j)$			
							{1,1}	{1,2}	{2,0}	{2,1}	{2,2}
c_1	{1, 2}	0	0	1	1	1	1	1	1	1	1
c_2	{1, 3}	1	1	1	1	1	1	1	1	1	1
c_3	{1, 4}	3	2	1	1	1	1	1	1	1	1
c_4	{2, 3}	2	3	1	1	1	1	1	1	1	1
c_5	{2, 4}	4	4	1	1	1	1	1	1	1	1
c_6	{3, 4}	5	5	1	1	1	1	1	1	1	1

Algorithm 1: SA, sequential algorithm to verify a CA.

```

1  $t\_wise(\mathcal{M}_{N,k}, v, t, \mathcal{K}_l, \mathcal{K}_u)$ 
2 begin
3    $Miss \leftarrow 0, iMax \leftarrow t - 1, P \leftarrow 0$ 
4   /* Get initial t-tuple */
5    $J \leftarrow getInitialTuple(k, t, c_{\mathcal{K}_l})$ 
6   while  $J_t \leq k$  and  $f(J) \leq \mathcal{K}_u$  do
7     covered  $\leftarrow 0$ 
8     for  $n \leftarrow 1$  to  $N$  do
9        $P_{f(J),g(\mathcal{M}_{n,J})} \leftarrow P_{f(J),g(\mathcal{M}_{n,J})} + 1$ 
10      for  $j \leftarrow 1$  to  $v^t$  do
11        if  $P_{f(J),j} > 0$  then
12          covered  $\leftarrow$  covered + 1
13       $Miss \leftarrow Miss + v^t - covered$ 
14      /* Calculates the next t-tuple */
15       $J_t \leftarrow J_t + 1$ 
16      if  $J_t > k$  and  $iMax > 0$  then
17         $J_{iMax} \leftarrow J_{iMax} + 1$ 
18        for  $i \leftarrow iMax + 1$  to  $t$  do
19           $J_i \leftarrow J_{i-1} + 1$ 
20        if  $J_{iMax} > k - t + iMax$  then
21           $iMax \leftarrow iMax - 1$ 
22        else
23           $iMax = t$ 
24 return Miss

```

We propose the Algorithm 2 as a method that generates c_F , according to a lexicographical, without generating its previous t -tuples c_i , where $i < F$. To explain the purpose of the Algorithm 2, let's consider the $CA(9; 2, 4, 3)$ shown in Figure 1. This CA has as set \mathcal{C} the elements found in column 1 of Table 4a. The algorithm `getInitialTuple` with input $k = 4$, $t = 2$, $F = 3$ must return $J = \{1, 4\}$, i.e. the values of the t -tuple c_3 . The Algorithm 2 is optimized to find the vector $J = \{J_1, J_2, \dots, J_t\}$ that corresponds to F . The value J_i is calculated according to

$$J_i = \min_{j \geq 1} \left\{ \Delta_i \leq \sum_{l=J_{i-1}+1}^j \binom{k-l}{t-i} \right\}$$

where

$$\Delta_i = F - \sum_{m=1}^{i-1} \sum_{l=J_{m-1}+1}^{J_m-1} \binom{k-l}{t-m}$$

and

$$J_0 = 0.$$

Algorithm 2: Get initial t -tuple to PA.

```

1  $getInitialTuple(k, t, c_i)$ 
2 Output: Initial t-tuple each core
3 begin
4    $\Theta \leftarrow i, iK \leftarrow 1, iT \leftarrow 1$ 
5    $kint \leftarrow \binom{k-iK}{t-iT}$ 
6   for  $i \leftarrow 1$  to  $t$  do
7     while  $\Theta > kint$  do
8        $\Theta \leftarrow \Theta - kint$ 
9        $iK \leftarrow iK + 1$ 
10       $kint \leftarrow \binom{k-iK}{t-iT}$ 
11       $J_i \leftarrow iK$ 
12       $iK \leftarrow iK + 1$ 
13       $iT \leftarrow iT + 1$ 
14       $kint \leftarrow \binom{k-iK}{t-iT}$ 
15 return J

```

In conclusion, the Algorithm 2 only requires the computation of $O(t \times k)$ binomials to compute the n initial t -tuples of the PA. This represents a great improvement in contrast with the naive approach that would require the generation of all the $\binom{k}{t}$ t -tuples, as done in the SA. The next section presents the experimental results of using SA and PA approaches proposed in this paper applied to the verification of CAs.

4. Experimental design to compare the different approaches to verify a CA

This section presents an experimental design and results derived from testing the approaches described in the last section. The purpose of the experiment is to show the performance of each approach and the limitations that one can find on them. The two approaches were implemented in C language. SA was compiled with `gcc -O3` and PA was compiled with `mpicc -O3`. SA and PA were run in the Tirant supercomputer belonging to the Spanish Supercomputing Network [28].

In order to show the performance of the verification of CAs algorithm, two experiments were developed. The first experiment uses a benchmark that contains 16 binary CAs, the second experiment uses a benchmark that contains 16 ternary CAs. Each benchmark was created using the following parameters: $t = \{2, 3, 4\}$, $k = \{64, 128, 256, 512\}$. We developed a specific implementation of Galois algorithm [11], enhanced by the use of logarithmic tables [29].

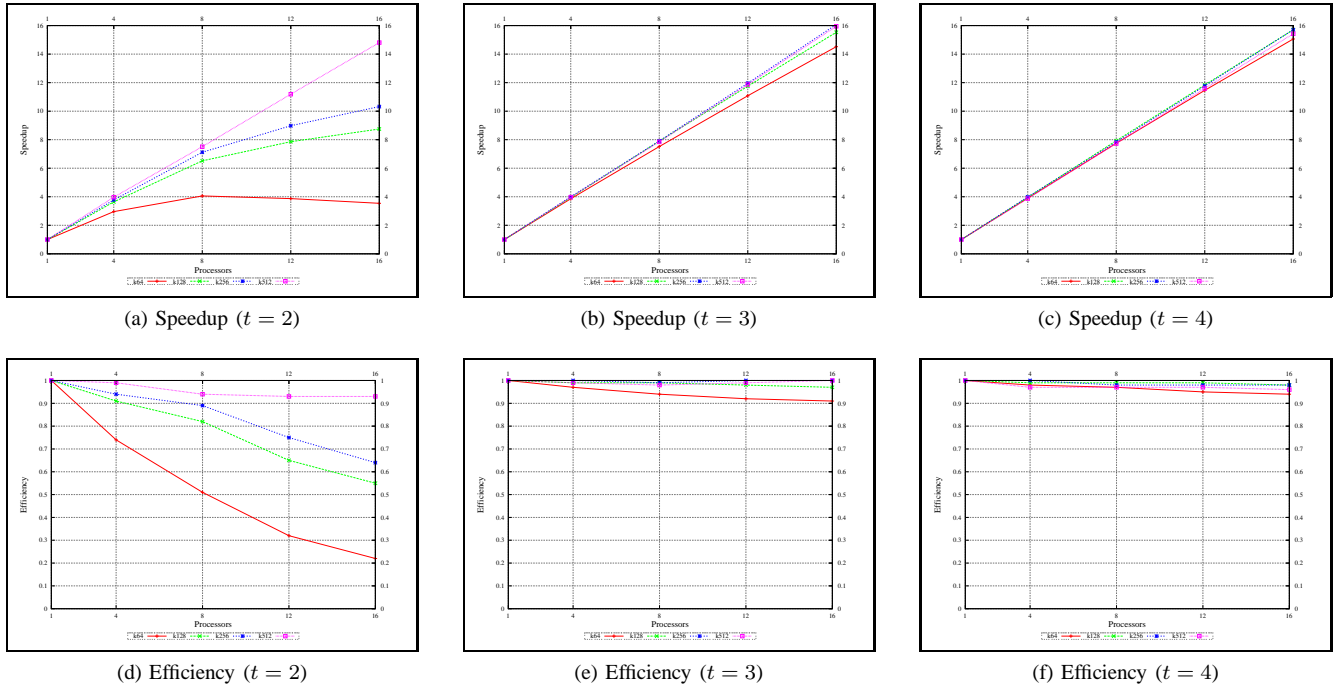


Fig. 2: Performance analysis when $t = \{2, 3, 4\}$ $k = \{64, 128, 256, 512\}$ and $v = 2$.

We use this implementation to generate the 32 CAs. For simplicity, the only multi-core approach considered during the comparison was PA having a maximum number of cores of 16 and the maximum time was 48 hours.

The comparison was based on the speedup (\mathcal{S}) and the efficiency (\mathcal{E}) on a parallel architecture against a single core case. We define speedup as the factor by which the execution time for the application changes, $\mathcal{S}(n) = \frac{\tau(s)}{\tau(n)}$, where τ_s is execution time on a single processor and τ_n is execution time on a multiprocessor. Efficiency gives fraction of time that processors are being used on computation, ($\mathcal{E} = \frac{\mathcal{S}(n)}{n}$).

4.1 Binary benchmark

This subsection presents a comparison between the SA and PA approaches using a binary benchmark with the purpose of studying the variation in the performance between a single core approach and an approach with more than one core. Table 5 shows the time (in seconds) spent by SA and PA to verify the experimental CAs. The first four columns describe the test case, the rest of the columns show the time spent by SA and PA, respectively, to verify each instance.

Figures 2a and 2d shows the performance achieved for the instances of strength $t = 2$; it is possible to observe that for almost all the cases the speedup is almost linear up to 4 cores, and after that it drops considerably. Figures 2b and 2e shows the performance for the instances where $t = 3$. A linear speedup up to 8 cores, and after that it drops slightly. Figures 2c and 2f shows the performance for the instances

Table 5: Comparison of the performance of SA and PA. This table shows the time (in seconds) spent by each approach when verifying the chosen benchmark.

N	t	k	v	SA	PA (16 cores)
64	2	64	2	0.000382	0.000108
64	3	64	2	0.018693	0.001287
64	4	64	2	0.662790	0.044016
64	5	64	2	14.490000	0.911106
128	2	128	2	0.001592	0.000182
128	3	128	2	0.159930	0.010300
128	4	128	2	12.760802	0.812173
128	5	128	2	709.999744	47.536213
256	2	256	2	0.005984	0.000580
256	3	256	2	1.276129	0.079378
256	4	256	2	206.547729	13.151242
256	5	256	2	27240.528490	1753.620271
512	2	512	2	0.024898	0.001681
512	3	512	2	10.938686	0.685710
512	4	512	2	3568.107378	231.240136
512	5	512	2	898675.740682	69418.805718

where $t = 4$. The performance shows a better behavior in the speedup and the efficiency than the ones observed for the cases where $t = \{2, 3\}$, i.e. the speedup is almost linear and the efficiency is almost 100% in all the instances showed.

4.2 Ternary benchmark

This subsection presents a comparison between the SA and PA approaches using a benchmark that contains 16 ternary CAs. The Table 6 shows the time (in seconds) spent by SA and PA to verify the experimental CAs. The first four columns describe the test case, the rest of the columns show

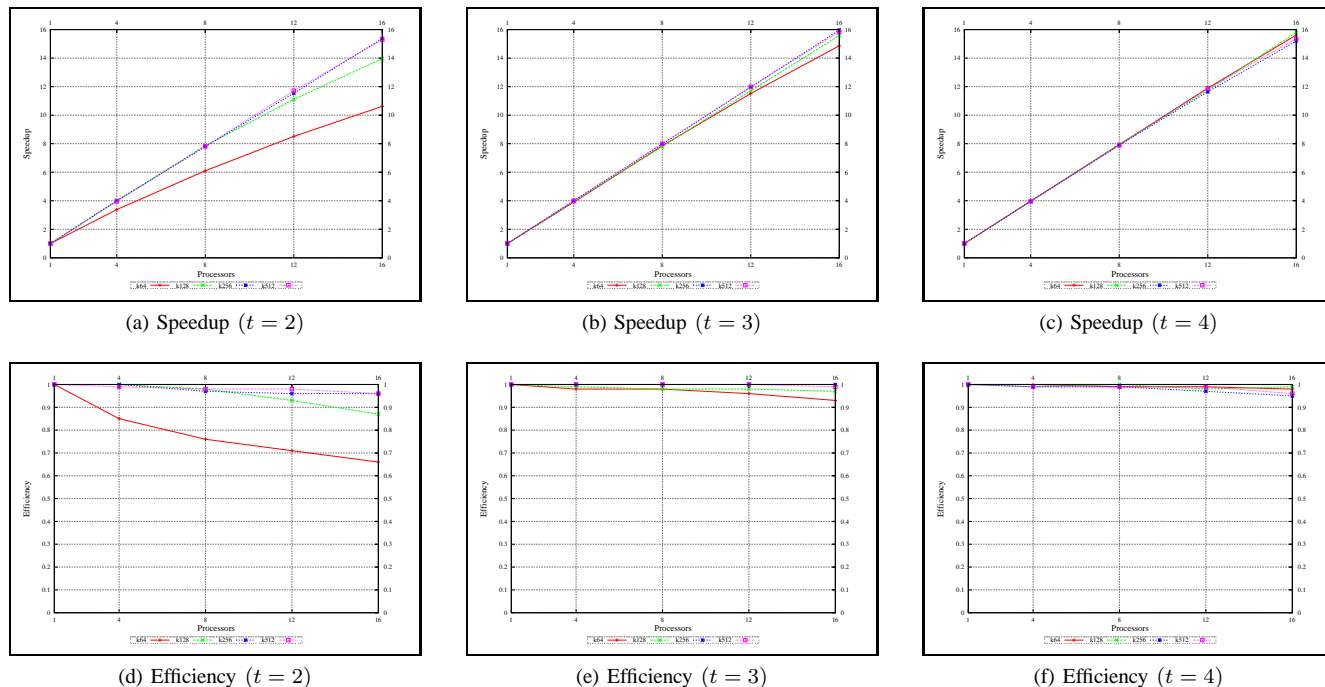


Fig. 3: Performance analysis when $t = \{2, 3, 4\}$, $k = \{64, 128, 256, 512\}$ and $v = 3$.

the time spent by SA and PA, respectively, to verify each instance. A cell in the table with the expression **n.a.** denotes that this experiment does not end at the maximum available time (48 hours).

Table 6: Comparison of the performance of SA and PA. This table shows the time (in seconds) spent by each approach when verifying the chosen benchmark.

N	t	k	v	SA	PA (16 cores)
64	2	64	3	0.000430	0.000255
64	3	64	3	0.104173	0.009200
64	4	64	3	2.233600	0.143011
64	5	64	3	35.201100	2.217308
128	2	128	3	0.007564	0.000590
128	3	128	3	1.109700	0.071200
128	4	128	3	68.076500	4.308390
128	5	128	3	2134.342164	135.955665
256	2	256	3	0.026556	0.001728
256	3	256	3	9.335492	0.584722
256	4	256	3	1895.894700	124.724641
256	5	256	3	132781.989216	9024.639374
512	2	512	3	0.112200	0.007339
512	3	512	3	80.039744	5.056195
512	4	512	3	41956.982600	2733.268051
512	5	512	3	n.a.	n.a.

To illustrate the scalability of our parallel algorithm we use the cases when $t = \{2, 3, 4\}$, $k = \{64, 128, 256, 512\}$ and $processors = \{4, 8, 12, 16\}$. The Figure 3 shows that the acceleration is almost linear and the efficiency is about 95% when $v = 3$. Therefore we can conclude that our parallel algorithm scales very well.

5. Conclusions

This paper presents sequential and parallel approaches to solve the problem of verifying CAs. The approaches were used to verify a benchmark formed by 32 matrices which were proposed to be verified as CAs. The CAs to be verified have alphabets $v = \{2, 3\}$, strengths $t = \{2, 3, 4, 5\}$ and number of columns $k = \{64, 128, 256, 512\}$.

The results of the two experiments presented showed SA as a good option when $t \leq 3$; however, this approach consumed a lot of time when $t \geq 4$ and $k > 128$ which made it impractical to use.

We proposed an optimized algorithm to calculate the initial t -tuple, its computational cost is $O(t \times k)$. PA approach worked well in the cases when $t > 2$. In general, the results showed that the partition strategy proposed for PA was efficient for the CA verification problem due to it required an almost null communication; this fact is observed in the linear speedup seen experimentally. Therefore we can conclude that our parallel algorithm scales very well.

Finally, given that the results gave birth to new CAs, another contribution of this paper is a new set of CAs which have been concentrated in the repository available at <http://www.tamps.cinvestav.mx/~jtj/CA.php>. In this web site it is possible to download the CAs, with better upper bounds than the ones given at the NIST Web site [30]. The current best upper bounds can be found in [31].

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] 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 '03)*. IEEE Computer Society, 2003, pp. 394–405.
- [2] R. Kuhn, Y. Lei, and R. Kacker, "Practical combinatorial testing: Beyond pairwise," *IT Professional*, vol. 10, no. 3, pp. 19–23, 2008.
- [3] J. N. Cawse, *Experimental Design for Combinatorial and High Throughput Materials Development*, J. N. Cawse, Ed. John Wiley & Sons, Inc., 2003.
- [4] A. Hedayat, N. Sloane, and J. Stufken, *Orthogonal Arrays: Theory and Applications*, A. Hedayat, N. Sloane, and J. Stufken, Eds. Springer-Verlag, 1999.
- [5] D. E. Shasha, A. Y. Kouranov, L. V. Lejay, M. F. Chou, and G. M. Coruzzi, "Using combinatorial design to study regulation by multiple input signals: A tool for parsimony in the post-genomics era," *Plant Physiol*, vol. 127, no. 4, pp. 1590–1594, 2001.
- [6] M. S. Phadke, *Quality Engineering Using Robust Design*, M. S. Phadke, Ed. Prentice Hall PTR, 1995.
- [7] K. K. Vadde and V. R. Syrotiuk, "Factor interaction on service delivery in mobile ad hoc networks," *IEEE J Sel Area Comm*, vol. 22, no. 7, pp. 1335 – 1346, 2004.
- [8] K. Burr and W. Young, "Combinatorial test techniques: Table-based automation, test generation and code coverage," in *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*. West, 1998, pp. 503–513.
- [9] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE T Software Eng*, vol. 32, no. 1, pp. 20–34, 2006.
- [10] D. J. Kleitman and J. Spencer, "Families of k-independent sets," *Discrete Math*, vol. 6, no. 3, pp. 255–262, 1973.
- [11] K. A. Bush, "Orthogonal arrays of index unity," *Ann Math Stat*, vol. 23, no. 3, pp. 426–434, 1952.
- [12] Y. Lei and K. Tai, "In-parameter-order: A test generation strategy for pairwise testing," in *The 3rd IEEE International Symposium on High-Assurance Systems Engineering*, ser. HASE '98. IEEE Computer Society, 1998, pp. 254–261.
- [13] C. J. Colbourn, "Combinatorial aspects of covering arrays," *Le Matematiche*, vol. 58, pp. 121–167, 2004.
- [14] M. Chateaneuf and D. L. Kreher, "On the state of strength-three covering arrays," *J Comb Des*, vol. 10, no. 4, pp. 217–238, 2002.
- [15] K. Meagher and B. Stevens, "Group construction of covering arrays," *Journal of Combinatorial Designs*, vol. 13, no. 1, pp. 70–77, 2005.
- [16] M. B. Cohen, C. J. Colbourn, and A. C. H. Ling, "Constructing strength three covering arrays with augmented annealing," *Discrete Math*, vol. 308, no. 13, pp. 2709–2722, 2008.
- [17] A. Hartman and L. Raskin, "Problems and algorithms for covering arrays," *Discrete Mathematics*, vol. 284, no. 1-3, pp. 149 – 156, 2004.
- [18] 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.
- [19] Y. Tung and W. S. Aldiwan, "Automating test case generation for the new generation mission software system," in *Aerospace Conference Proceedings*. IEEE, 2000, vol. 1, pp. 431–437.
- [20] R. C. Bryce and C. J. Colbourn, "The density algorithm for pairwise interaction testing," *Softw Test Verif Rel*, vol. 17, no. 3, pp. 159–182, 2007.
- [21] 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 '07)*. IEEE, 2007, pp. 549–556.
- [22] M. Younis, K. Zamli, and N. Mat Isa, "IRPS: An Efficient Test Data Generation Strategy for Pairwise Testing," in *Knowledge-Based Intelligent Information and Engineering Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, vol. 5177, pp. 493–500.
- [23] J. Torres-Jimenez and E. Rodriguez-Tello, "Simulated annealing for constructing binary covering arrays of variable strength," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–8.
- [24] J. Martinez-Pena, J. Torres-Jimenez, N. Rangel-Valdez, and H. Avila-George, "A heuristic approach for constructing ternary covering arrays using trinomial coefficients," in *IBERAMIA*, ser. Lecture Notes in Computer Science, A. F. K. Morales and G. R. Simari, Eds., vol. 6433. Springer, 2010, pp. 572–581.
- [25] T. Berling and P. Runeson, "Efficient evaluation of multifactor dependent system performance using fractional factorial design," *IEEE T Software Eng*, vol. 29, no. 9, pp. 769–781, 2003.
- [26] L. Gonzalez-Hernandez, N. Rangel-Valdez, and J. Torres-Jimenez, "Construction of mixed covering arrays of variable strength using a tabu search approach," in *COCOA (1)*, ser. Lecture Notes in Computer Science, W. Wu and O. Daescu, Eds., vol. 6508. Springer, 2010, pp. 51–64.
- [27] E. Rodriguez-Tello and J. Torres-Jimenez, "Memetic algorithms for constructing binary covering arrays of strength three," in *Artificial Evolution 2009*. Springer, 2009, vol. 5975, pp. 86–97.
- [28] Tirant supercomputer, "Spanish Supercomputing Network," URL: <http://www.uv.es/siuv/cas/zcalculo/res/descrpcion.wiki>. Accessed 20 April 2011.
- [29] J. Torres-Jimenez, N. Rangel-Valdez, A. L. Gonzalez-Hernandez, and H. Avila-George, "Construction of logarithm tables for Galois Fields," *International Journal of Mathematical Education in Science and Technology*, vol. 42, no. 1, pp. 91–102, 2010.
- [30] National Institute of Standards and Technology, "NIST covering array tables," URL: <http://math.nist.gov/coveringarrays/>. Accessed 20 April 2011.
- [31] C. J. Colbourn, "Covering array tables for t=2,3,4,5,6," URL: <http://www.public.asu.edu/~ccolbou/src/tabby/catable.html>. Accessed 20 April 2011.

Methodology for Performance Evaluation of the Input/Output System

Sandra Méndez¹, Dolores Rexachs¹, and Emilio Luque¹

¹Computer Architecture and Operating System Department (CAOS)
Universitat Autònoma de Barcelona, Bellaterra, Barcelona, Spain

Abstract—*The increase of processing units, the speed and the computing power, and the complexity of scientific applications that use high performance computing require more efficient Input/Output systems. To use the I/O system efficiently it is necessary to know its performance capacity to determine whether it fulfills I/O requirements of the applications. This paper proposes a methodology for I/O system performance evaluation for computers clusters. The approach encompasses the characterization of the application, I/O system, and devices. We select and evaluate the I/O factors that have an impact on the application performance by considering the application and the I/O architecture configuration. During the analysis of the I/O configuration, we identify configurable factors that impact the I/O system performance. In the evaluation phase, we evaluate the application under different I/O configurations and we determine the inefficiency by analyzing the difference between measured values and characterized values.*

Keywords: Parallel I/O System, I/O Architecture, I/O Configuration, I/O Characterization

1. Introduction

The increase of processing units, the advance in speed and computing power, and the increasing complexity of scientific applications that use high performance computing require more efficient Input/Output(I/O) Systems. Due to the historical “gap” between the computing performance and I/O performance, in many cases, the I/O system becomes the bottleneck of the parallel systems. The efficient use of the I/O system and the identification of I/O factors that influence the performance can help to cover this “gap”. To use the I/O system efficiently, it is first necessary to know its performance capacity to determine if it fulfills the application I/O requirements. There are several papers on performance evaluation of I/O system and since I/O system performance depends on the software and hardware of the I/O, these studies are made for specific parallel computer configurations. Roth [1] presented event tracing for characterizing the I/O demands of applications on the Jaguar Cray XT supercomputer. Fahey [2] experimented in the I/O system of the Cray XT, and the analysis was focused in the LUSTRE filesystem. Laros [3] carried out a performance evaluation of

I/O configuration. Previous papers do not directly consider the I/O characteristics of applications.

We propose the I/O system evaluation by analyzing each level on the I/O path. Furthermore, we take into account the application I/O requirements and the I/O architecture configuration. The proposed methodology has three phases: characterization, the analysis of I/O configuration, and the evaluation. In the characterization phase, we extract the I/O requirements of the application, bandwidth and IOPs (I/O operations per second) at filesystem level, interconnection network, I/O library and I/O devices. In the analysis of the I/O configuration phase we identify configurable factors that impact the performance of the I/O system. We analyze the filesystem, I/O node connection, placement and state of buffer/cache, data redundancy and service redundancy. We use these factors along with application behavior to compare and analyze existent I/O configurations in the cluster. In the evaluation phase we collect metrics of the application execution under different configurations. The inefficiency is determined by analyzing the difference between measured values and characterized values.

The rest of this article is organized as follows: Section II introduces our proposed methodology. In Section III we review the experimental validation of this proposal. Finally, in the last section, we present conclusions and future work.

2. Proposed Methodology

The methodology (Fig. 1) has three phases: characterization, I/O configuration analysis and evaluation. The I/O system performance is related to I/O time, which depends on the transfer rate and I/O operations. We analyze the application and I/O system to identify the possible points of inefficiency. We also extract information in order to select the more suitable configuration for the application.

2.1 Characterization

We analyze the application I/O characteristics and the I/O system. Fig. 2 shows information obtained in this phase. Here we describe the process of the characterization.

2.1.1 Parallel I/O System

Parallel system was characterized at I/O library level, filesystem (local, distributed and/or parallel) and storage

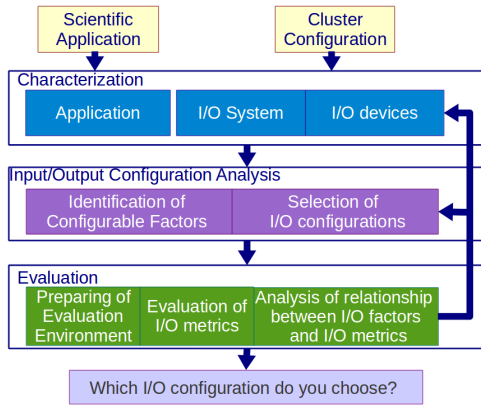


Fig. 1: Methodology for Performance Evaluation of I/O System

Application characterization	System characterization	Devices characterization
What? •Access Patterns •Workload •Number of files •Number of processes •Data mapping •I/O library	Bandwidth and latency IO operations for: I/O Library Local Filesystem Distributed Filesystem Parallel Filesystem Communication network I/O network	Bandwidth and latency I/O operations for: Disk RAID SSD
How? •Tracing Tools: mpe, DARSHAN Dynamically Linked Libraries (LD_PRELOAD), PAS2P •Linux System tools (vmstat) •Application Documentation	•Filesystem benchmarks (bonnie++, iozone, IOR) •I/O Library benchmarks (b_eff_io, IOR)	•Disk benchmarks (bonnie++, iozone) •System Tools (sar, hdparm, iostat)

Fig. 2: Characterization phase

network (Direct Attached Storage (DAS), Network Attached Storage (NAS), Storage Area Network (SAN) and NAS devices (NASD)). We characterize the IOPs and bandwidth for each level, as shown in figure 3. To evaluate filesystem, IOzone [4] and/or bonnie++ [5] benchmarks can be used. For library level it is possible to use b_eff_io [6] or IOR [7].

The I/O system analyzed corresponds to the cluster Aohyper. This cluster has the following characteristics: 8 nodes AMD Athlon(tm) 64 X2 Dual Core Processor 3800+, 2GB RAM memory, 150GB local disk, and local filesystem linux ext4; 1 NFS server with RAID 1 (2 disks) with 230GB capacity and RAID 5 (3 disks) with stripe=256KB and 917GB capacity, both with write-cache enabled (write back); two Gigabit Ethernet networks, one for communication and one for data. The parallel system and storage devices characterization were done with IOzone (TABLE 1). In this case the minimum bandwidth corresponds to I/O library. The experiments were performed at block level with a file size which doubles the memory size. Block size was changed from 32KB to 16MB. For I/O library, the IOR benchmark was used wich was configured for 8 segments, from 16MB to 128MB block size and transfer block size from 32 to 256KB. It was launched with 8 and 16 processes. evaluation

2.1.2 Scientific Application

We extract the type, amount and size of I/O operation at library level. TABLE 2 shows the application I/O requirements. This information is used in the evaluation phase to

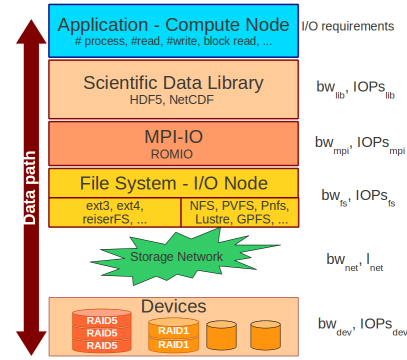


Fig. 3: I/O system Characterization

Table 1: I/O system characterization of Aohyper - Average (MB/sec)

Measures	read home	write home	read raid1	write raid1	read raid5	write raid5
I/O Library	27	48	28	48	29	48
Local Filesystem	68	85	133	113	273	193
Global Filesystem	47	50	48	50	47	50

determine whether application performance is limited by the application or by the I/O system. To evaluate the application characterization at process level, an extension of PAS2P [8] tracing tool was developed. To expand the characterization phase, the methodology is applied to Block Tridiagonal(BT) application of NAS Parallel Benchmark suite (NPB)[9]. NAS BT-IO is a compute and I/O intensive application with communications between processes. Therefore, compute node, networks and I/O system have an influence on applications performance. The TABLE 2 shows the characterization done for the class C of NAS BT-IO in full and simple subtype.

2.2 Input/Output Configuration Analysis

Users do not normally know I/O system in detail. They trust that the parallel system is capable to processing all I/O requests and hope to recover processed data when they demand. We propose the idea that an application can use a different set of I/O configurable resources of existent I/O architecture. This configurable resource set is called I/O configurations. We evaluate I/O system components for evaluating the impact on application performance due to configuration, as shown in Fig. 3.

Table 2: Characterization NAS BT-IO - Class C - 16 processes

Parameters	full	simple
number of application files	1	1
number of reads	640	2,073,600 and 2,125,440
number of writes	640	2,073,600 and 2,125,440
block size to read	10 MB	1.56KB and 1.6KB
block size to write	10 MB	1.56KB and 1.6KB
number of open calls	32	32
number of close calls	32	32
number of processes	16	16

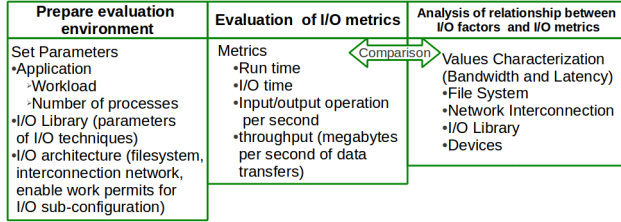


Fig. 4: Evaluation Phase

2.2.1 Identification of Configurable Factors

We considered the following configurable factors in the I/O architecture: filesystem type and placement (local, distributed and parallel), number and type of interconnection network (dedicated and shared to compute), state and placement of buffer/cache, RAID level, number and placement of I/O node. The cluster Aohyper has two RAID levels: 1 and 5 and JBOD (Just a Bunch Of Disks). There is no redundancy of service. Aohyper has an NFS server that is also I/O node for sharing accesses. For local accesses there are eight I/O nodes and the data sharing must done by the user. At filesystem level, the cluster has ext4 as its local filesystem and NFS as its global filesystem. There are two networks, one for services and other for data transfer.

2.2.2 Selection of I/O Configurations

The configuration is selected in function of performance provided in I/O path and the RAID level. Three configurations were selected: JBOD, with RAID1 and with RAID5. However, the I/O system also has configurations with software RAID levels composed by local disks and with one or two networks for communication and data.

2.3 Evaluation

The Fig. 4 shows the evaluation step. In this phase, we prepared the evaluation environment, we defined I/O metrics for evaluation and we analyzed application performance in different configurations.

2.3.1 Preparation the Evaluation Environment

For the evaluation phase we set parameters of application, library and architecture. For our example, we evaluate the NAS BT-I/O class C with MPICH library.

2.3.2 Evaluation of I/O metrics

The metrics for the evaluation were: execution time, I/O time (time to do reading and writing operations), I/O operations per second (IOPs), latency of I/O operations and throughput (number of megabytes transferred per second).

2.3.3 Analysis of relationship between I/O factors and I/O metrics

We compared the measures of the application execution with characterized values on I/O path levels by each con-

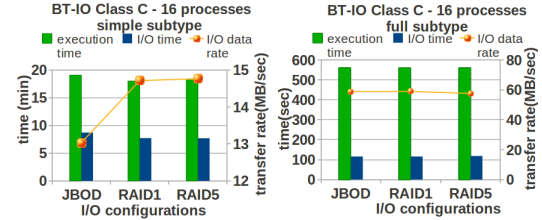


Fig. 5: NAS BT-I/O Class C 16 Processes

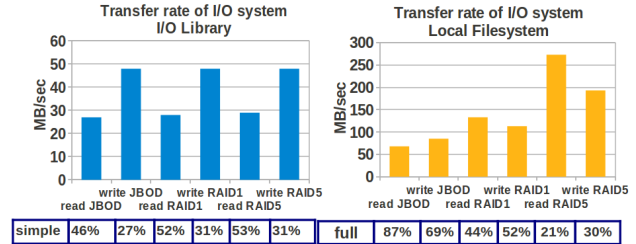


Fig. 6: Transfer rate percentages for NAS BT-I/O

figuration. Each configuration has associated an array $A = [bw_{lib}, bw_{mpi}, bw_{fs}, bw_{net}, bw_{disk}]$, each value corresponds to a level in path I/O. We have compared the array A with the application transfer rate, taking into account the I/O operation type. With this information, we can identify possible points of inefficiency in the I/O path. When the throughput of application is much lower than $\min(A)$ then the application does not efficiently use the capacity of I/O system due to its access patterns.

Fig. 5 shows the execution time, I/O time and throughput for NAS BT-I/O class C using 16 processes executed in the cluster Aohyper on three I/O configurations. The evaluation is for full subtype (with collectives I/O) and simple (without collectives). In Fig. 6, we observed a low transfer rate, NAS BT-I/O simple subtype use about 50% of I/O library characterized values for reading and about 30% for writing in the three configurations. NAS BT-I/O simple subtype does 4,199,040 writes and 4,199,040 reads with block size of 1,600 and 1,640 bytes (TABLE 2). This has a high penalization in I/O time impacting on execution time (Fig. 5).

Fig. 6 shows NAS BT-I/O full subtype (with block size 10 MB for reading and writing, TABLE 2), the rate transfer is about 70% of local filesystem characterized values for writing and 90% for reading in JBOD. In RAID1, the rate transfer is about 50% of local filesystem characterized values for writing and 45% for reading. The configuration with RAID 5 has higher performance for simple subtype. However, this I/O capacity is not exploited (Fig. 6) by simple subtype due to its access pattern.

3. Experimentation

In order to test the methodology, an evaluation of NAS BT-I/O for 16 and 64 processes in a different cluster was done. Such a cluster is called cluster A. It is composed

Table 3: I/O system of Cluster A

Measures in MB/sec	read raid5	write raid5
I/O library	32	51
Local Filesystem	260	200
Global Filesystem	89	79

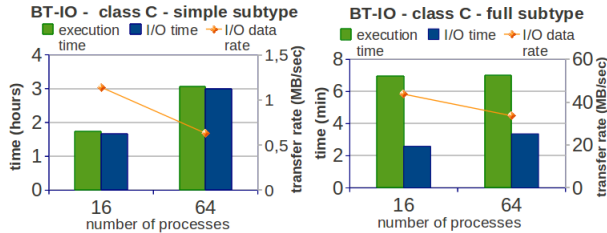


Fig. 7: NAS BT-IO Class C - 16 and 64 processes

of 32 compute nodes: 2 x Dual-Core Intel (R) Xeon (R) 3.00GHz, 12 GB of RAM, and 160GB SATA disk Dual Gigabit Ethernet. A front-end node as NFS server: Dual-Core Intel (R) Xeon (R) 2.66GHz, 8 GB of RAM, 5 of 1.8 TB RAID and Dual Gigabit Ethernet. Characterization of I/O system of cluster A is presented in TABLE 3. IOR tests to evaluate the library of I/O were done with 8 segments, block size of 256MB, and 256KB transfer block.

Characterization of NAS BT-IO is similar to the cluster Aohyper. The characterization of application with 16 processes in cluster A is shown in TABLE 2. As we analyze application behavior, it is not necessary to characterize the application again in other systems for the same class and number of processes. Fig. 7 shows execution time, I/O time and throughput for NAS BT-IO full subtype and simple. NAS BT-IO simple subtype (Fig. 8) for 16 processes has a transfer rate of about 3% of characterized value on I/O library for writing and 2% for reading operations. The transfer rate is lower for 64 processes with 1% for reading and about 2% for writing. The simple subtype of NAS BT-IO is limited by I/O in this I/O configuration of cluster A. We analyze the full subtype in (Fig. 8), where we see the full subtype surpasses the barrier of the I/O library. The Fig. 8 shows percentages of characterized values for NFS. The higher transfer rate is achieved with 16 processes and the lowest I/O time impacting in execution time. With this information we can observe that full subtype of NAS BT-IO is limited by computing or communication in Cluster A. NAS BT-IO full subtype does not achieve 60% of NFS characterized values

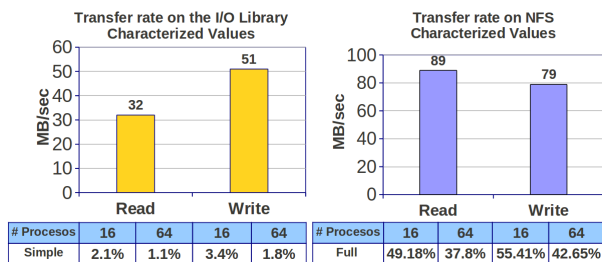


Fig. 8: Transfer rate percentage NAS BT-IO

and the execution time is increased with more processes due to the increase of the I/O operations and communication among processes.

4. Conclusion

A methodology to analyze I/O performance of parallel computers was shown. Such methodology encompasses the characterization of the I/O system at different levels: devices, I/O system and application. The configuration of different elements that impact on performance were analyzed and evaluated by considering the application and the I/O architecture. This methodology is applied in two different clusters for the NAS BT-IO benchmark. I/O systems features are shown, as well as their impact on the applications performance.

An application I/O model is being defined to give support to the evaluation and design of configurations. This model is based on the characteristics of the application and the I/O system. The objective of this model is to determine which configuration of I/O meets the user's performance requirements, taking into account the behavior of the application I/O in a given system. For other configurations evaluation, we are analyzing the simulation framework SIMCAN[10]. We plan to use this tool to model full I/O architectures.

Acknowledgment

This research has been supported by MICINN-Spain under contract TIN2007-64974.

References

- [1] P. C. Roth, "Characterizing the i/o behavior of scientific applications on the cray xt," in *PDSW '07: Procs of the 2nd int. workshop on Petascale data storage*. USA: ACM, 2007, pp. 50-55.
- [2] M. Fahey, J. Larkin, and J. Adams, "I/o performance on a massively parallel cray xt3/xt4," in *Parallel and Distributed Procs, 2008. IPDPS 2008. IEEE Int. Symp. on*, 14-18 2008, pp. 1-12.
- [3] 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.
- [4] W. D. Norcott, "Iozone filesystem benchmark," Tech. Rep., 2006. [Online]. Available: <http://www.iozone.org/>
- [5] R. Coker, "Bonnie++ filesystem benchmark," Tech. Rep., 2001. [Online]. Available: <http://www.coker.com.au/bonnie++/>
- [6] R. Rabenseifner and A. E. Koniges, "Effective file-i/o bandwidth benchmark," in *Euro-Par '00: Procs from the 6th Int. Euro-Par Conference on Parallel Procs*. London, UK: Springer-Verlag, 2000, pp. 1273-1283.
- [7] S. J. Shan, Hongzhang, "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/
- [8] 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.
- [9] 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.
- [10] A. Núñez, *et al.*, "Simcan: a simulator framework for computer architectures and storage networks," in *Simutools '08*. Belgium: ICST, 2008, pp. 1-8.

Computational Aspects of Silicate Networks

Paul Manuel¹, Indra Rajasingh², Albert William² and Antony Kishore²

¹ Department of Information Science, Kuwait University, Kuwait 13060

² Department of Mathematics, Loyola Collage, Chennai, India, 600 034

Abstract - In this paper we consider a new interconnection network motivated by molecular structure of a chemical compound SiO_4 . The different forms of silicates available in nature lead to the introduction of the silicate networks. We investigate various parameters related to this network. We provide an addressing scheme for the nodes of the network and also an algorithm that enables us to draw the silicate network in the two dimensional plane aesthetically. Certain properties of this network are brought out using classical geometry. We also identify edge disjoint cycles of this network. The embedding of the honeycomb and hexagonal networks into silicates is studied in order to demonstrate that they are all computationally equivalent.

Keywords: Silicate networks, topological properties, mesh-like architectures, embedding, and drawing algorithm

1 Introduction

Interconnection network is a programmable system that transports data between terminals. Early work on interconnection networks was motivated by the needs of the communications industry, particularly in the context of telephone switching. With the growth of the computer industry, applications for interconnection networks within computing machines began to become apparent. As interest in parallel processing grew, a large number of networks were proposed for processor to memory and processor to processor interconnection [10].

An *interconnection network* consists of a set of processors, each with a local memory, and a set of bidirectional links that serve for the exchange of data between processors. A convenient representation of an interconnection network is by an undirected graph $G = (V, E)$ where each processor is a vertex in V and two vertices are connected by an edge if and only if there is a direct communication link between processors [10].

A few networks such as Hexagonal, Honeycomb, and grid networks, for instance, bear resemblance to atomic or molecular lattice structures. *Honeycomb networks*, built recursively using the hexagon tessellation [14,15,16], are widely used in computer graphics [11], cellular phone

base station [12], image processing [2], and in chemistry as the representation of benzenoid hydrocarbons [15] and Carbon Hexagons of Carbon Nanotubes [9]. *Hexagonal networks* are based on triangular plane tessellation, or the partition of a plane into equilateral triangles [3,12,16]. Hexagonal network represents a host cyclotrimer with halogenated monocarbaborane anions [1] and Silicon Carbide [13]. *Carbon nanotubes* consist of shells of sp^2 -hybridized carbon atoms forming a hexagonal network, arranged helically within a tubular motif [1].

In this paper, we deal with silicate networks which was introduced recently; silicates are obtained by fusing metal oxides or metal carbonates with sand. Essentially all the silicates contain SiO_4 tetrahedra. The corner vertices of SiO_4 tetrahedron represent oxygen ions and the center vertex represents the silicon ion. Graph theoretically, we call the corner vertices as *oxygen nodes* and the center vertex as *silicon node*. See Figure 1. The minerals are obtained by successively fusing oxygen nodes of two tetrahedra of different silicates.

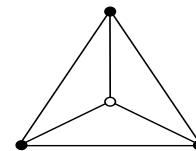


Figure 1: SiO_4 tetrahedra where the corner vertices represent oxygen ions and the center vertex the silicon ion

In this paper, we study the topological properties of silicate networks as it has been studied for other interconnection networks [2,3,5,8,10,14,16]. We study its structure and properties from the perspective of computer science. In order to compare the computational powers of silicate networks with the other similar mesh-like architectures, those architectures are embedded into silicates. We also look at the compound from the perspectives of chemistry. We study the topological structure of silicates. We identify an equilateral property of silicates and we partition the oxygen edges into edge disjoint cycles. We propose an addressing scheme of silicate networks and map the nodes of silicate networks onto a Cartesian plane to draw the silicate network aesthetically.

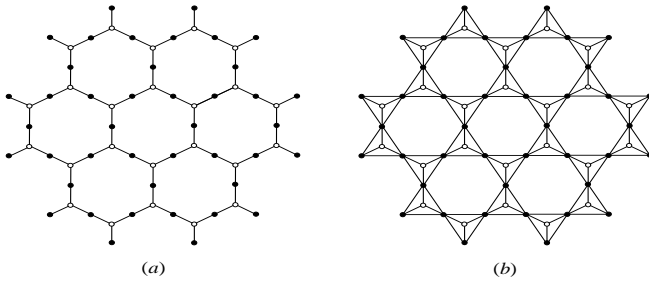


Figure 2: Construction of Silicate Network $SL(n)$ from honeycomb network $HC(n)$

2 Properties of Silicate Networks

A silicate network can be constructed in different ways. Here in this paper we describe the construction of a silicate network from a honeycomb network. Consider a honeycomb network $HC(n)$ of dimension n . Place silicon ions on all the vertices of $HC(n)$. Subdivide each edge of $HC(n)$ once. Place oxygen ions on the new vertices. Introduce $6n$ new pendant edges one each at the 2-degree silicon ions of $HC(n)$ and place oxygen ions at the pendent vertices. See Figure 2(a). With every silicon ion associate the three adjacent oxygen ions and form a tetrahedron as in Figure 2(b). The resulting network is a silicate network $SL(n)$. The parameter n of $SL(n)$ is called the *dimension* of $SL(n)$. The graph in Figure 2(b) is a silicate network of dimension 2.

Theorem 1: The number of nodes in $SL(n)$ is $15n^2 + 3n$. The number of edges of $SL(n)$ is $36n^2$. \square

When all the silicon nodes are deleted from a silicate network, we obtain a new network which we shall call as an *Oxide Network*. An n -dimensional oxide network is denoted by $OX(n)$.

Theorem 2: The number of nodes in $OX(n)$ is $9n^2 + 3n$. The number of edges of $SL(n)$ is $18n^2$. \square

3 Addressing the nodes of Silicate Networks

In order to study the properties of silicate networks, it is important to assign a unique identity *id* (coordinate) to each node of silicate network. First we shall propose a coordinate system that can be used to assign an *id* to each node of oxide network. Then we shall extend this coordinate system to silicate network.

We shall adapt the coordinate system that was proposed for a honeycomb network by Stojmenovic [14] or a hexagonal network by Nocetti et al. [12]. Three axes, α, β and γ parallel to three edge directions and at mutual angle of 120 degrees between any two of them are introduced, as indicated in Figure 3. The three coordinate axes are $\alpha = 0, \beta = 0,$ and $\gamma = 0$ respectively. We call lines parallel to the coordinate axes as α -lines, β -lines and γ -lines. Here $\alpha = h$ and $\alpha = -k$ are α -lines on either side of α -axis.

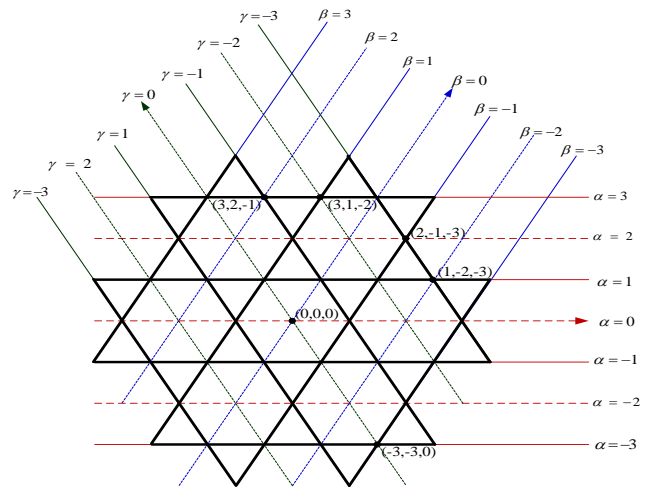


Figure 3: Coordinate System in Oxide Networks

Lemma 1: In (α, β, γ) coordinate system, the three lines $\alpha = h, \beta = k,$ and $\gamma = l$ intersect if and only if $k = h + l$.

Proof:

Since the α -lines are parallel to X -axis, the α -line " $\alpha = 0$ " is taken as the x -axis. Line $\alpha = h, h \in \mathbb{Z}$ of $OX(n)$ is mapped to $y = h$ in the Cartesian system. A β -line $\beta = k$ makes an angle 60° with X -axis and forms a y -intercept $2k$ in the Cartesian system. Thus line $\beta = k, k \in \mathbb{Z}$ is mapped to $y = (\tan 60^\circ)x + 2k$ which is $y = \sqrt{3}x + 2k$. A γ -line $\gamma = \ell$ makes an angle 120° with X -axis and forms a y -intercept -2ℓ in the Cartesian system. Thus line $\gamma = \ell, \ell \in \mathbb{Z}$ is mapped to $y = (\tan 120^\circ)x - 2\ell$ which is $y = -\sqrt{3}x - 2\ell$. See Figure 3.

The lines $\alpha = h, \beta = k,$ and $\gamma = l$ intersect

\Leftrightarrow the third line passes through the point of intersection of the first two lines

$\Leftrightarrow y = h$ passes through the point of intersection of $y = \sqrt{3}x + 2k$ and $y = -\sqrt{3}x - 2\ell$.

$\Leftrightarrow y = h$ is satisfied by $\left(-\frac{(k+l)}{\sqrt{3}}, k-l\right)$

$\Leftrightarrow h = k - \ell$ i.e., $k = h + \ell$. \square

Lemma 2: (α, β, γ) represents an oxide molecule if and only if

- (1) $\beta = \alpha + \gamma$ and
- (2) at least one of α, β, γ is odd.

Proof: The proof is obvious. See Figure 3. \square

The above lemma can also be restated as

A node of $OX(n)$ is assigned a triple (a, b, c) when the node is the intersection of lines $\alpha = a, \beta = b,$ and $\gamma = c$ and at least one of a, b, c is odd.

Remark: In an oxide molecule, exactly two of α, β, γ must be odd.

Each silicon node is at the centroid of three oxygen nodes of a tetrahedral SiO_4 . Thus it is enough to specify an addressing scheme for the oxygen nodes of $SL(n)$. For all practical purposes, we don't need separate *id* for silicon

nodes since the silicate network is completely characterized by the oxide network. However for the sake of completeness, one can assign *ids* to silicon nodes by applying the formula of centroid of a triangle.

4 The edge set of $OX(n)$ is partitioned into edge disjoint cycles

Definition: An edge of $OX(n)$ is called α -edge if it is in some α -line. A β -edge and a γ -edge are defined in the same way. A cycle of $OX(n)$ is said to be a *symmetric cycle* if it is formed by an α -edge, a β -edge and a γ -edge alternatively. Notice that the number of edges of any symmetric cycle of $OX(n)$ is a multiple of 3. We demonstrate that the edge set of $OX(n)$ is partitioned into edge-disjoint symmetric cycles.

Theorem 3: The edge set of $OX(n)$ can be partitioned into edge-disjoint symmetric cycles.

Proof: As a strategy to partition the edge set of $OX(n)$, we orient the edges of $OX(n)$ as follows:

Orient the edges of lines $\alpha = 1, 3 \dots 2n + 1$ in the positive direction.

Orient the edges of lines $\alpha = -1, -3 \dots -2n - 1$ in the negative direction.

Orient the edges of lines $\beta = 1, 3 \dots 2n + 1$ in the positive direction.

Orient the edges of lines $\beta = -1, -3 \dots -2n - 1$ in the negative direction.

Orient the edges of lines $\gamma = 1, 3 \dots 2n + 1$ in the positive direction.

Orient the edges of lines $\gamma = -1, -3 \dots -2n - 1$ in the negative direction.

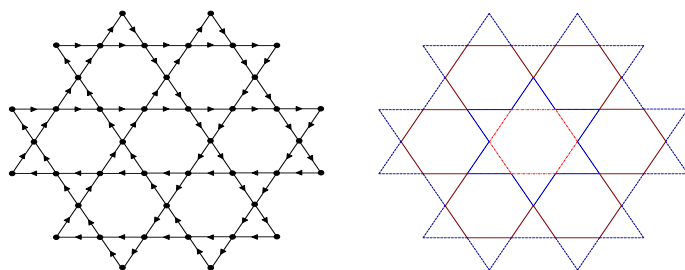


Figure 4: Edge set of $OX(n)$ is partitioned into cycles

See the orientation in Figure 4. Symmetric cycles in the oriented $OX(n)$ are unique. Hence it is now easy to partition $OX(n)$ into edge-disjoint symmetric cycles. The innermost symmetric cycle is a hexagon of the length 6. Then onwards, each successive symmetric cycle forms a layer on the previous one. See Figure 4. The length of each symmetric cycle is a multiple of 6. \square

5 Equilateral Triangle Property of Silicate Network

Three vertices u, v, w of a graph $G(V, E)$ are said to form an *equilateral triangle* if $d(u, v) = d(v, w) = d(w, u)$ where $d(x, y)$ denotes the distance between x and y . There is an interesting equilateral triangular property of silicate networks.

Theorem 4: Three vertices $A(x_1, x_2, x_3), B(y_1, y_2, y_3)$ and $C(z_1, z_2, z_3)$ of $SL(n)$ form an equilateral triangle if $x_1 = y_1, y_2 = z_2$ and $z_3 = x_3$. \square

Continuing the above theorem, we discuss a stronger result. Consider a triangle ABC formed by some α -line, β -line and γ -line. By the above theorem, ΔABC is equilateral. Let $a_1, a_2 \dots a_r$ be the nodes on the β -line between B and C . Let $b_1, b_2 \dots b_r$ be the nodes on the α -line between C and A . Let $c_1, c_2 \dots c_r$ be the nodes on the γ -line between A and B . See Figure 5. We know that $d(A, B) = d(A, C)$. The interesting observation is that $d(A, B) = d(A, a_i) = d(A, C)$ for $i = 1, 2 \dots r$.

Theorem 5: Let ΔABC denote a triangle of $SL(n)$ formed by three vertices $A(x_1, x_2, x_3), B(y_1, y_2, y_3)$ and $C(z_1, z_2, z_3)$ such that $x_1 = y_1, y_2 = z_2$ and $z_3 = x_3$. Let a be a node on the β -line between B and C , let b be a node on the α -line between C and A and let c be a node on the γ -line between A and B . Then $d(A, B) = d(B, C) = d(C, A) = d(A, a) = d(B, b) = d(C, c)$. \square

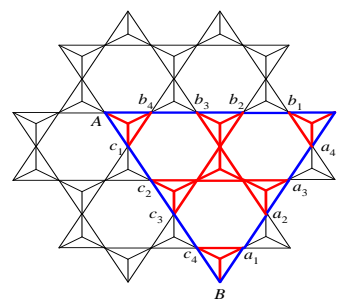


Figure 5: Equilateral triangle property

6 Embedding of Honeycomb and Hexagonal Networks in Silicate Networks

Let G and H be finite graphs with n vertices. $V(G)$ and $V(H)$ denote the vertex sets of G and H respectively. $E(G)$ and $E(H)$ denote the edge sets of G and H respectively. An embedding [10] f of G into H is defined as follows:

- f is a bijective map from $V(G) \rightarrow V(H)$.
- f is a one-to-one map from $E(G)$ to $\{P_f(f(u), f(v)) / P_f(f(u), f(v))$ is a path in H between $f(u)$ and $f(v), (u, v) \in E(G)\}$.

The dilation $\hat{D}_f(G, H)$ of an embedding f of G into H is defined as

$$\hat{D}_f(G, H) = \max_{(u,v) \in E(G)} |P_f(f(u), f(v))|$$

where $|P_f(f(u), f(v))|$ denotes the length of the path $P_f(f(u), f(v))$.

Then, the dilation of G into H is defined as

$$\hat{D}(G, H) = \min_f \hat{D}_f(G, H)$$

where the minimum is taken over all embeddings f of G into H . The dilation problem for a graph G into H is that of finding an embedding of G into H that induces the dilation $\hat{D}(G, H)$.

We next state the results pertaining to embedding of honeycomb and hexagonal networks into silicate networks.

Theorem 6: The dilation of the embedding of a honeycomb network of dimension n into a silicate network of the same dimension is 2.

Proof: From the structure of $SL(n)$ described in Section 3, it follows that the subdivision of the honeycomb network of dimension n is a subgraph of $SL(n)$. See Figure 6. Hence the dilation of the embedding of $HC(n)$ into $SL(n)$ is 2. \square

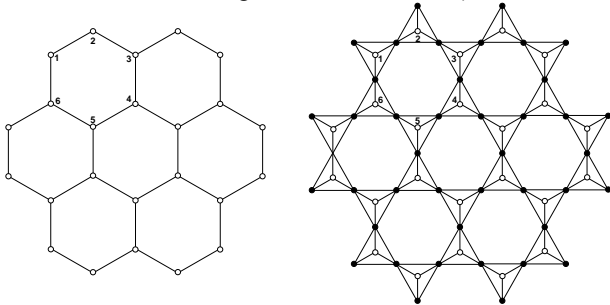


Figure 6: Dilation of the embedding is 2

Theorem 7: The dilation of the embedding of $HX(n)$ into $SL(n - 2)$ is at most 3.

Proof: We provide an embedding of $HX(n)$ into $SL(n - 2)$ as follows:

Step 1: The nodes of $HX(n)$ are labeled along α -lines from left to right as in Figure 7.

Step 2: Each hexagon of $HX(n)$ between successive $\alpha = 2k + 1$ and $\alpha = 2k + 3$ lines is mapped into the corresponding hexagon of $SL(n - 2)$. See Figure 8.

Step 3: The center node of a hexagon of $HX(n)$ is mapped into a node which sits on top of the corresponding hexagon of $SL(n - 2)$. See Figure 9.

The Figure 10 shows the embedding of $HX(n)$ into $SL(n - 2)$. It is easy to verify that the dilation of this embedding is 3. \square

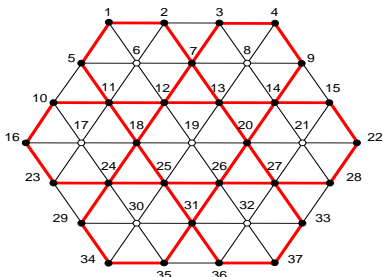


Figure 7: The nodes of Hexagon $HX(n)$ are labeled

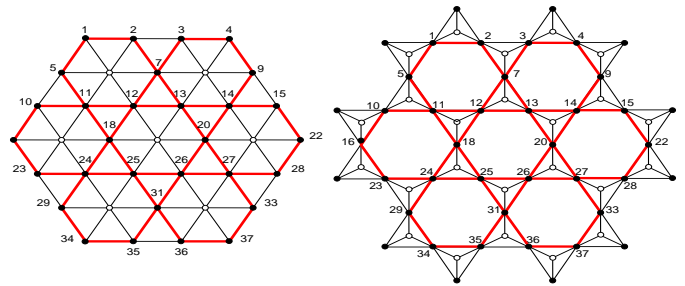


Figure 8: Hexagons of $HX(n)$ between successive $\alpha = 2k + 1$ and $\alpha = 2k + 3$ lines are mapped into the corresponding hexagons of $SL(n - 2)$.

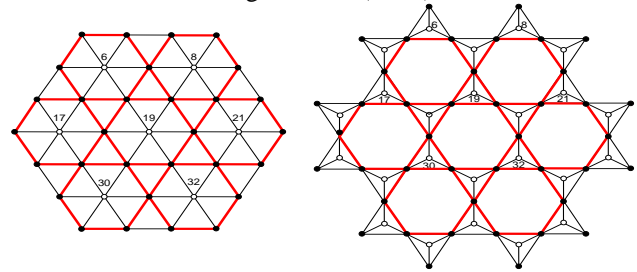


Figure 9: Center node of the hexagon of $HX(n)$ is mapped into a node which sits on top of the corresponding hexagon of $SL(n - 2)$

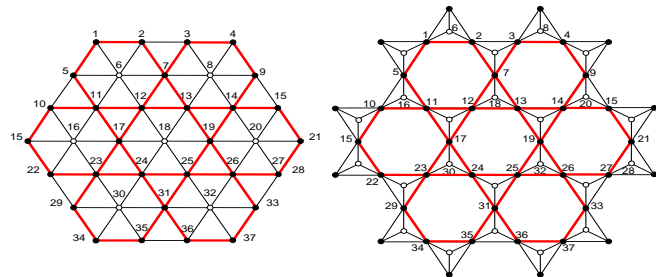


Figure 10: An embedding of $HX(n)$ into $SL(n - 2)$. The edge (6, 11) of $HX(n)$ is dilated along 6, 1, 5, 11 in $SL(n - 2)$. The dilation of $HX(n)$ into $SL(n - 2)$ is 3.

Thus algorithms such as minimum communication cost, routing, and broadcasting algorithms of honeycomb and hexagon networks [5, 8, 10] can be simulated in silicate networks with a time complexity that differ by a constant.

7 More Properties of Silicate Networks

Theorem 8: (i) $OX(n)$ contains a Hamiltonian path.
(ii) $SL(n)$ contains a Hamiltonian path.

Proof: (i) By the method of induction.

$OX(1)$ contains a Hamiltonian path. See Figure 11 (a).

Assume the theorem is true for $OX(k)$. i.e., $OX(k)$ has a Hamiltonian path.

To prove the theorem is true for $OX(k + 1)$.

By theorem 3, The edge set of $OX(n)$ is partitioned into edge-disjoint symmetric cycles. By induction, there exists a Hamiltonian path in $OX(k)$. This path is connected to the outermost symmetric cycle thus producing a Hamiltonian path in $OX(k+1)$. See Figure 11 (b)

(ii) The proof of (ii) is similar to the above proof as the path $v_1 v_2 v_3$ is replaced by $v_1 v_2 v_4 v_3$. See Figure 12 (a) and 12 (b). \square

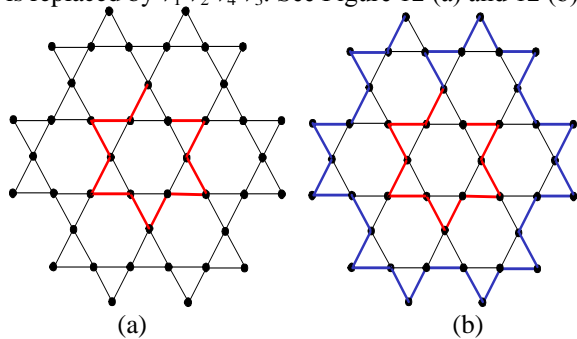


Figure 11: Hamiltonian path in $OX(1)$, and $OX(2)$

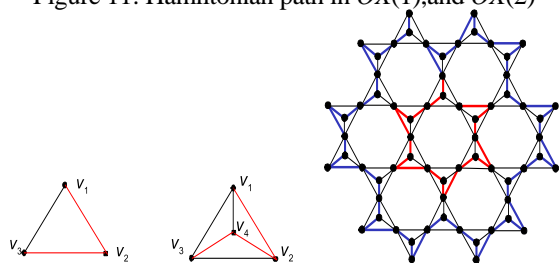


Figure 12: (a) A Hamiltonian path in a triangle of $OX(n)$,
 (b) A Hamiltonian path inside a triangle of $SL(n)$,
 (c) A Hamiltonian path in $SL(2)$.

Remark: Using exhaustive MATLAB simulation, we observe that $OX(n)$ has no Hamiltonian cycle. However there is no mathematical proof to show that $OX(n)$ has no Hamiltonian cycle. It is interesting to derive a logical proof to show that $OX(n)$ has no Hamiltonian cycle.

Theorem 9:

- (i) $OX(n)$ is tripartite and its chromatic number is 3.
- (ii) $SL(n)$ is 4-partite and its chromatic number is 4.

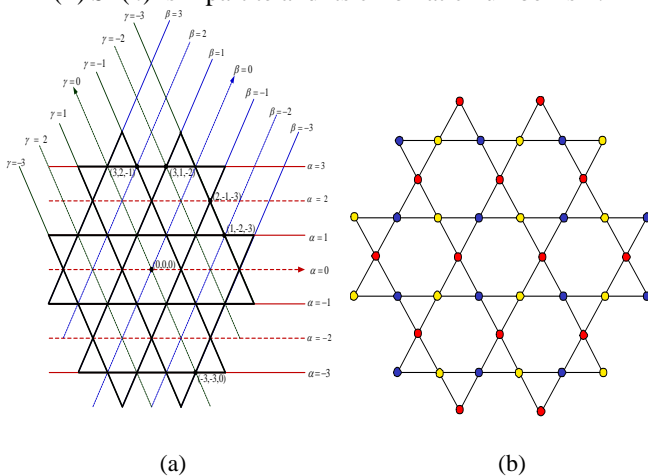


Figure 13: (a) Coordinate System in Oxide Networks,
 (b) Chromatic coloring for $OX(2)$

Proof: Claim that the chromatic number $\chi(OX(n))$ is 3.

See Figure 13. Label the nodes on the α - lines corresponding to $\alpha = \dots -4, -2, 0, 2, 4 \dots$ by color 1 (red). Label the nodes on the α - lines corresponding to $\alpha = \dots -3, -1, 1, 3, 5 \dots$ by color 2 (yellow) and color 3 (blue) such that adjacent nodes do

not have the same color. Since no two nodes on $\alpha = \dots -4, -2, 0, 2, 4 \dots$ are adjacent and each node on these α -lines is adjacent to nodes colored yellow and blue, $\chi(OX(n)) \leq 3$. Since the chromatic number of a triangle is 3, $\chi(OX(n)) \geq 3$. Thus $\chi(OX(n)) = 3$ and hence $OX(n)$ is tripartite. \square

Remark:

- (i) $SL(n)$ is not bipartite since it contains odd cycles.
- (ii) $OX(n)$ is Eulerian but $SL(n)$ is not Eulerian since it contains odd degree vertices.

Domestic Number: A dominating set in a graph is a set of vertices such that every vertex in the graph is either in the set or has a neighbour in the set. A domestic partition is a partition of the vertices so that each part is a dominating set of the graph. The domestic number of a graph G denoted by $D(G)$ is the maximum number of dominating sets in a domestic partition of the graph G , or equivalently, the maximum number of disjoint dominating sets[4]. The domestic partition problem is that of partitioning the vertices of a graph into the maximum number of disjoint dominating sets. The domestic partition problem is one of the classical NP – hard problems [7].

Lemma 1 [4]: Every graph G satisfies $D(G) \geq 1$, and unless G contains an isolated node, $D(G) \geq 2$.

Lemma 2 [4]: Let δ denote the minimum degree of a node in the graph G . Then $D(G) \leq \delta + 1$.

Proof: Since the node with minimum degree must have some neighbor (or itself) in each of the disjoint dominating sets, $D(G) \leq \delta + 1$. \square

Theorem 10: (i) The domestic number of $OX(n)$ is 3.
 (ii) The domestic number of $SL(n)$ is 4.

Proof: By Lemma 2, $D(OX(n)) \leq 3$ and $D(SL(n)) \leq 4$.

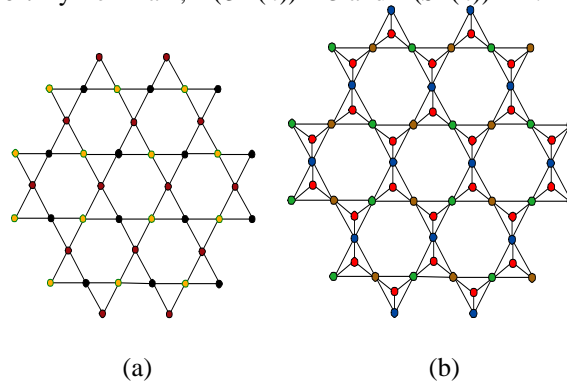


Figure 14: (a) Domestic coloring for $OX(2)$,
 (b) Domestic coloring for $SL(2)$

We have presented in Figure 14, a domestic partition of size 3 for $OX(n)$ and size 4 for $SL(n)$. Therefore $D(OX(n)) \geq 3$ and $D(SL(n)) \geq 4$. \square

8 Drawing Algorithm for Silicate Networks

Let us classify the edges of a tetrahedral SiO_4 as follows: An edge incident at a silicon node is called *silicon edge*. Otherwise it is called an *oxygen edge*. The drawing algorithm

provides a method to draw $SL(n)$ in the Cartesian plane. That is, it provides a formula to map a node (α, β, γ) of $SL(n)$ into a point (x, y) of Cartesian System. Our objective is to draw $SL(n)$ in Cartesian plane in such a way that all the drawn edges of $SL(n)$ are of equal length in the 2-dimensional plane. However, it is not possible to draw a tetrahedral SiO_4 (a complete graph on 4 vertices) in the 2-dimensional plane such that all edges are of equal length. Therefore, we design a drawing algorithm such that all the silicon edges are of the same length and all the oxygen edges are of the same length. As mentioned earlier, it is enough to design a drawing algorithm for oxide network.

The oxygen edges of a tetrahedral SiO_4 form a triangle. In other words, these three oxygen edges should make an equilateral triangle in order to be of equal length. Geometrically, these three edges make an angle of 60° with each other. Moreover, these three oxygen edges are on α -lines, β -lines, and γ -lines of $OX(n)$ respectively. In order to keep all the oxygen edges of equal length, α -lines, β -lines, and γ -lines of $OX(n)$ are drawn as follows:

All α -lines are parallel to X -axis, β -lines make 60° with X -axis and γ -lines make 120° with X -axis.

Successive α -lines (β -lines, and γ -lines) are equally spaced in the Cartesian plane.

(1) Since the α -lines are parallel to X -axis, the α -line " $\alpha = 0$ " is taken as the x -axis. Line $\alpha = (2h+1)$, $h \in Z$ of $OX(n)$ is mapped to $y = (2h+1)$ in the Cartesian system.

(2) A β -line $\beta = (2k+1)$ makes an angle 60° with X -axis and forms a y -intercept $2(2k+1)$ in the Cartesian system. See Figure 3. Thus line $\beta = (2k+1)$, $k \in Z$ is mapped to $y = (\tan 60^\circ)x + 2(2k+1)$ which is $y = \sqrt{3}x + 2(2k+1)$.

(3) A γ -line $\gamma = (2\ell+1)$ makes an angle 120° with X -axis and forms a y -intercept $-2(2\ell+1)$ in the Cartesian system. See Figure 3. Thus line $\gamma = (2\ell+1)$, $\ell \in Z$ is mapped to $y = (\tan 120^\circ)x - 2(2\ell+1)$ which is $y = -\sqrt{3}x - 2(2\ell+1)$.

From (1), we have

$$\alpha = (2h+1) \text{ and } y = (2h+1), h \in Z$$

Thus

$$y = \alpha. \quad (A)$$

From (2) and (3) we have

$$\begin{aligned} \beta &= (2k+1) \text{ and } y = \sqrt{3}x + 2(2k+1). \\ \gamma &= (2\ell+1) \text{ and } y = -\sqrt{3}x - 2(2\ell+1). \end{aligned}$$

Solving the above two equations, we get

$$x = -(\beta + \gamma)/\sqrt{3}. \quad (B)$$

Combining (A) and (B), we arrive at a function f that maps a node (α, β, γ) of $OX(n)$ to a node of Cartesian System as follows:

$$f(\alpha, \beta, \gamma) = (-(\beta + \gamma)/\sqrt{3}, \alpha). \quad (C)$$

This function f provides an algorithm to draw $OX(n)$ in a Cartesian plane.

Once the oxide network is drawn in the Cartesian plane, placing silicon nodes is rather simple. As we know, a silicon node is at the centroid of three oxygen nodes of a tetrahedral SiO_4 . If (x_1, y_1) , (x_2, y_2) , (x_3, y_3) are the Cartesian coordinates of oxygen nodes of a tetrahedral SiO_4 , then $((x_1+x_2+x_3)/3, (y_1+y_2+y_3)/3)$ is the Cartesian coordinate of the silicon node

of the tetrahedral SiO_4 . This completes the drawing algorithm of $SL(n)$.

Theorem 11: A silicate network can be drawn in a two-dimensional Cartesian plane such that all the silicon edges are of equal length and all the oxygen edges are of equal length. □

8.1 MATLAB program to draw $SL(n)$

```
function silicate(n)
close all;axis square;hold on
T=0:0.01:1;
rad1=0.07;
for i=-2*n:2*n
    for j=-2*n:2*n
        for k=-2*n:2*n
            if (j-k == i && (mod(i,2)~=0 || mod(j,2)~=0 ||
                mod(k,2)~=0))
                fill(-(j+k)/sqrt(3)+rad1*cos(2*pi*T),
                    i+rad1*sin(2*pi*T),'b')
            if (mod(i,2)==0 && i~=2*n)
                for t = -1:1
                    if t~=0
                        fill(-(j+1/3+k-1/3)/sqrt(3)+rad1*cos(2*pi*T),
                            t*(i+2/3)+rad1*sin(2*pi*T),'r')
                        plot([-j+1/3+k-1/3)/sqrt(3) -(j+k)/sqrt(3)],
                            [(i+2/3) i], 'b')
                        plot([-j+1/3+k-1/3)/sqrt(3) -(j+1+k)/sqrt(3)],
                            [(i+2/3) i+1], 'b')
                        plot([-j+1/3+k-1/3)/sqrt(3) -(j+k-1)/sqrt(3)],
                            [(i+2/3) i+1], 'b')
                        plot([-j+1/3+k-1/3)/sqrt(3) -(j+k)/sqrt(3)],
                            [-(i+2/3) -i], 'b')
                        plot([-j+1/3+k-1/3)/sqrt(3) -(j+1+k)/sqrt(3)],
                            [-(i+2/3) -(i+1)], 'b')
                        plot([-j+1/3+k-1/3)/sqrt(3) -(j+k-1)/sqrt(3)],
                            [-(i+2/3) -(i+1)], 'b')
                    end
                end
            end
        end
    end
end
for i=1:n
    for j = -1:1
        if j~=0
            plot([(4*(n-1)+3-2*(i-1))/sqrt(3) -(4*(n-1)+3-2*(i-1))/sqrt(3)],
                [j*(2*i-1) j*(2*i-1)])
            plot([j*(4*(n-1)+3-2*(i-1))/sqrt(3) j*(2*(n-1)-4*(i-1))/sqrt(3)],
                [-(2*i-1) 2*n])
            plot([j*(4*(n-1)+3-2*(i-1))/sqrt(3) j*(2*(n-1)-4*(i-1))/sqrt(3)],
                [(2*i-1) -2*n])
        end
    end
end
hold off
```

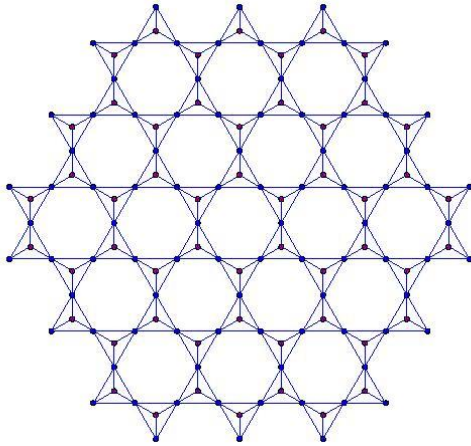


Figure 15: Silicate drawn using MATLAB

9 Conclusion

In this paper we have considered a new interconnection network motivated by the molecular structure of certain chemical compounds. We have investigated the topological and structural properties of this network. We have provided an addressing scheme for the nodes of the network and also an algorithm that enables us to draw the silicate network in the two dimensional plane aesthetically.

It is shown that any algorithms such as minimum communication cost, routing, and broadcasting algorithms of honeycomb and hexagon networks can be simulated in silicate networks with a time complexity by a difference of constant factors.

This paper is an eye opener for researchers in the sense that different networks can be derived using the ores and compounds available in nature.

10 References

- [1] Ahmad R, Franken A, Kennedy J. D, Hardie M. J, Group 1 Coordination Chains and Hexagonal Networks of Host Cyclotriveratrylene with Halogenated Monocarbaborane Anions, *Chemistry*, vol. 10(9):2190-8, May 3, 2004.
- [2] Bell S. B. M, Holroyd F. C, Mason D. C, A Digital Geometry for Hexagonal Pixels, *Image and Vision Computing*, vol. 7, pp 194-204, 1989.
- [3] Catherine Decayeux, David Seme, 3D Hexagonal Network: Modeling, Topological Properties, Addressing Scheme, and Optimal Routing Algorithm, *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 9, pp 875-884, 2005.
- [4] E.J. Cockayne and S. Hedetniemi, Towards a theory of domination in graphs. *Networks* 7 (1977), 247-261.
- [5] Day K, Tripathi A, A Comparative Study of Topological Properties of Hypercubes and Star Graphs, *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 1, pp 31-38, 1994.
- [6] Gamst A, Homogeneous Distribution of Frequencies in a Regular Hexagonal Cell System, *IEEE Transactions on Vehicular Technologies*, vol. 31, pp 132-144, 1982.
- [7] M. R. Garey and D. S. Johnson, Computers and Intractability: A guide to the Theory of NP – completeness. *Freeman*. 1979.
- [8] Hamid Reza Tajozakerin, Hamid Sarbazi-Azad, Enhanced-Star: A New Topology Based on the Star Graph, *LNCS*, vol. 3358, pp 1030-1038, 2005.
- [9] Hongwei Zhu, Kazutomo Suenaga, Jinqian Wei, Kunlin Wang, Dehai Wu, Atom-Resolved Imaging of Carbon Hexagons of Carbon Nanotubes, *J. Phys. Chem. C*, vol. 112, no. 30, pp 11098-11101, 2008.
- [10] Junming Xu, Topological Structure and Analysis of Interconnection Networks, *Kluwer Publishers*, 2001.
- [11] Lester L. N, Sandor J, Computer Graphics on Hexagonal Grid, *Computer Graphics*, vol. 8, pp 401-409, 1984.
- [12] Nocetti F.G, Stojmenovic I, Zhang J, Addressing and Routing in Hexagonal Networks with Applications for Tracking Mobile Users and Connection Rerouting in Cellular Networks, *IEEE Transactions On Parallel And Distributed Systems*, vol. 13, no. 9, pp 963-971, September 2002.
- [13] Paul Manuel and Indra Rajasingh, Minimum Metric Dimension of Silicate Networks, *Ars Combinatoria* (volume 98, January 2011).
- [14] Stojmenovic I, Honeycomb Networks: Topological Properties and Communication Algorithms, *IEEE Trans. Parallel and Distributed Systems*, vol. 8, pp 1036-1042, 1997.
- [15] Tosic R, Masulovic D, Stojmenovic I, Brunvoll J, Cyvin B. N, Cyvin S. J, Enumeration of Polyhex Hydrocarbons up to $h = 17$, *Journal of Chemical Information and Computer Sciences*, vol. 35, pp 181-187, 1995.
- [16] Wenjun Xiao, Behrooz Parhami, Further Mathematical Properties of Cayley Digraphs Applied to Hexagonal and Honeycomb Meshes, *Discrete Applied Mathematics*, vol. 155, no. 13, pp 1752-1760, 2007.

Improving Distributed Processing in the COPAR system

Stephen J. Hartley, Joel M. Crichlow, Michael Hosein

¹ Computer Science Dept, Rowan University, Glassboro, NJ 08028, hartley@elvis.rowan.edu

² Computer Science Dept, Rowan University, Glassboro, NJ 08028, crichlow@rowan.edu

³ Computing and Information Tech Dept, University of the West Indies, Trinidad, mhosein2006@gmail.com

Abstract - COPAR (Combining Optimism and Pessimism in Accessing Replicas) is a distributed transaction system that manages updates for widely-scattered nodes containing replicas of a database of resource counts. The distinguishing feature is that the nodes of the system are connected by a high-latency, low-bandwidth, or congested network. An example is the distribution of relief supplies in a large-scale disaster situation. To implement responses to allocation and deallocation requests at a particular node in a timely manner, the database of available resources is replicated at each node. This article describes the system architecture, and how it was modified to improve the performance.

Keywords: Distributed Transaction Processing, Replica Management, Concurrency Control, Availability, Optimistic Processing, Pessimistic Processing.

1 Introduction

Consider the following situation. A collection of travel agency offices is scattered over a large geographical area and connected by a slow or congested network. An airline flight with a certain number of available seats is scheduled to take off at a date and time in the future. Each office handles the flight reservation requests and cancellations of customers walking into or calling the office.

A similar situation is the following. A collection of dispersed disaster relief centers handles the requests for relief supplies by distressed people near each center. Supplies are delivered from a central warehouse to relief centers based on their need. The centers and a central warehouse are connected by a slow or congested network. Trucks will carry the requested supplies from the central warehouse to the relief centers in the near future.

In these and other similar situations, depending on the slow network to synchronize updates to a single shared database of available resources would result in unacceptably lengthy response times to transactions (allocation and deallocation requests). COPAR [1, 2, 3, 4, 5] is a system supporting the replication of the database of available resources at each node (travel agency, relief center) in order to shorten transaction response times.

2 COPAR Operation

Each node maintains two counts of available resources. One count is called the permanent or pessimistic count; the other count is called the temporary or optimistic count.

Changes to the permanent count are synchronized with all the other nodes over the slow network using the two-phase update/commit algorithm. Thus, this count is always identical at all the nodes and represents true resource availability on the airline flight or at the central relief supply warehouse. The temporary count is maintained separately and independently by each node.

In general, resource counts for availability are a single non-negative integer R , such as for airline seats, or a vector of non-negative integers (R_1, R_2, \dots, R_m) , such as for m types of relief supplies. Similarly, resource counts for transactions (allocation and deallocation requests) are a single integer r , negative for an allocation and positive for a deallocation or release, or a vector of integers (r_1, r_2, \dots, r_m) .

When the system is initialized, the permanent count P_j at each node j is set to the initial resource availability R_{initial} , such as 100 seats on an airplane or 200 first aid kits, 100 blankets, and 400 bottles of water at a central warehouse for disaster relief. The temporary count T_j at each node is set to the initial permanent count divided by the number of nodes n . T_j is then adjusted upward by an overbooking allowance c , called the cost bound, where $c \geq 1$.

$$T_j = cR_{\text{initial}} / n$$

For example, if there are four nodes, if R_{initial} is 100, and if c is 1.16, then P_j is set to 100 and T_j is set to 29 at each node.

Most reservation systems allow some overbooking to compensate for reservations that are not used, such as passengers not showing up for an airline flight or people not picking up supplies when delivered to a relief center. There is a cost of overbooking, though, such as compensating passengers denied boarding on an airline flight or worsening the situation of people needing relief supplies. Organizations using a reservation system must carefully evaluate the cost of overbooking and limit it to what can be afforded.

The system simulates a person walking into or calling a travel agency by generating a transaction r_i for $i = 1, 2, \dots$ and sending it to a node j . The m integers in a transaction are generated randomly and the node j is chosen at random from 1 to n . Transactions from the generator are numbered sequentially.

Each node maintains two queues of transactions, called the parent or owner queue and the child queue. A transaction coming into a node from the generator is added to the parent queue and is also broadcast to all nodes, including itself, to be added to each node's child queue. The transactions r_i in each node's parent queue are kept sorted in order of increasing i , in

other words, in the order generated by the transaction generator. Note that a particular transaction r_i is in exactly one node's parent queue.

Each node j has two processors (threads), one responsible for maintaining the parent queue and the permanent count P_j at the node, and the other responsible for maintaining the child queue and the temporary count T_j at the node.

The permanent processor at each node participates in a two-phase commit cycle with all the other node permanent processors. After the processing of transaction r_{i-1} by its parent, the node whose parent queue contains transaction r_i becomes the coordinator for the two-phase commit cycle that changes the permanent count P_j at all nodes j to $P_j + r_i$. If the number of resource types m is greater than one, the notation for this change is P_{jk} to $P_{jk} + r_{ik}$ for $k = 1, 2, \dots, m$. This change is subject to the restriction that $P_{jk} + r_{ik}$ is nonnegative for all k . If that is not the case, all P_j are left unchanged and the transaction r_i is marked as a violation.

At the end of the two-phase commit cycle, the owner (parent) of transaction r_i sends a message to all nodes, including itself, to remove r_i from the node's child queue if r_i is present.

The temporary processor at each node j removes the transaction r_i at the head of its child queue, if any, and calculates if r_i can be allocated or satisfied from its temporary (optimistic) count T_j . In other words, node j checks if it is the case that $T_{jk} + r_{ik}$ is non-negative for all $k = 1, 2, \dots, m$. If that is not the case, transaction r_i is discarded; otherwise, node j sets T_j to $T_j + r_i$ and sends a message to the parent (owner) node of the transaction, the node whose parent queue contains the transaction. When a node n receives such a message from node j for transaction r_i , node n makes two checks.

- Is this the first such message received from any node's temporary processor for transaction r_i ?
- Has transaction r_i been done permanently yet?

If this is not the first such message, a reply is sent to node j that it is not first and it should back out of the temporary allocation it did for r_i , that is, change its temporary count T_j back to $T_j - r_i$.

If this is the first such message, then if the transaction r_i has not yet been done permanently (pessimistically), node j sending the message is marked as the node having done transaction r_i temporarily (optimistically). If this is the first such message, but transaction r_i has already been done permanently, no node is recorded as having done the transaction temporarily.

When the permanent processor in a node j coordinates the two-phase commit for a transaction r_i and has decided that transaction r_i is a violation, that is, $P_k + r_{ik}$ is negative for one or more k , node j checks to see if the transaction was marked as having been done optimistically earlier by some node's temporary processor. If so, the transaction r_i is marked as "undone," meaning that a passenger with a reservation is denied boarding or a person promised relief supplies is not given any.

If no node has done the transaction optimistically and it is not a violation, the owner's temporary processor allocation T_j is "charged" for it, $T_j = T_j + r_i$. This is done to lessen the probability of a later transaction being performed optimistically but then marked "undone" by the permanent processor.

The goal of the COPAR system is that temporary processors will be able to generate a reservation for an airline flight or a promise for relief supplies more quickly than the permanent processors. Doing a transaction optimistically involves a pair of messages between two nodes, whereas the two-phase commit of pessimistic processing involves a message count proportional to the number of nodes in the system.

3 Working with stale information

In earlier versions of the COPAR system described in [1, 2, 3, 4], nodes that are lightly loaded (receive proportionately fewer transactions from the generator) or are on a faster segment of the network or have faster or multiple CPUs might respond more quickly to transactions temporarily. This faster response might reduce their temporary counts more quickly towards zero, while other nodes retain higher temporary counts, especially if there are overall more reservations (transactions requesting resources) than cancellations (transactions returning or releasing resources) in the simulation of a system, as would be expected in reserving airplane seats or distributing disaster relief supplies.

The earlier versions of the system attempted to respond to this by recalculating the temporary counts at the end of every two-phase commit permanent processing cycle, i.e.,

$$T_j = c P_j/n$$

where $c \geq 1$ is the overbooking allowance and n is the number of nodes.

A serious problem can occur if the permanent processors lag significantly behind the temporary processors. This might result if one of the nodes is connected to the others with a higher latency, lower bandwidth, or more congested segment of the network compared to the interconnections of the other nodes. The former node will rarely be the first to respond temporarily to transactions. However, it must participate in all two-phase commit cycles of the permanent processors, slowing them down.

Suppose there are overall more reservations than cancellations in the simulation of a system. The temporary processors of the nodes with fast interconnections will be able to respond quickly and will get ahead of the permanent processors. Each time the permanent processors calculate the new temporary counts of the nodes, they will be using the permanent count P_j resulting from processing a much older transaction than the temporary processors are currently processing.

The net effect is that the temporary processors will be able to reserve the same resource multiple times, resulting in a much higher overbooking than intended by the cost bound factor c and resulting in many more temporary transactions “undone” by the permanent processors when they finally later encounter the temporarily done transactions.

We see this happening in Table 1, which contains statistical performance data from [4]. The local nodes are on the same subnet of a local area network at Rowan University in New Jersey and the UWI node is at a remote location (University of West Indies, in the Caribbean island of Trinidad) connected to the Rowan nodes over a wide area network (the Internet).

200 Transactions 1 per sec	Done temporarily	Violations	Undone
5 local nodes	95	69	3
4 local nodes + UWI	200	69	69
200 Transactions 5 per sec	Done temporarily	Violations	Undone
3 local nodes	102	69	6
2 local nodes + UWI	191	69	64

Table 1: Original results before the fix

4 Improving performance

We removed the step in the permanent processor after the two-phase commit that calculates the new values of the temporary counts T_j from the permanent counts P_j .

We see the improvement in Table 2. In simulations involving a remote node, there are many fewer temporarily done transactions that must be marked “undone” by the permanent processors.

200 Transactions 1 per sec	Done temporarily	Violations	Undone
5 local nodes	133	69	16
4 local nodes + UWI	137	69	15
200 Transactions 5 per sec	Done temporarily	Violations	Undone
3 local nodes	130	69	10
2 local nodes + UWI	136	69	14

Table 2: Results after the fix

5 Further research

We are investigating ways to deal with the following possibly occurring while the system handles transactions. Nodes that are lightly loaded (receive proportionately fewer transactions from the generator) or are on a faster segment of the network or have faster or multiple CPUs might respond more quickly to transactions temporarily. If there are more reservations than cancellations over time, this faster response might reduce their temporary counts more quickly towards zero, while other nodes retained higher temporary counts.

One proposed attempt to respond to this is by shifting temporary counts away from the nodes with higher counts to nodes with lower counts, with the idea that since they were able to respond more quickly temporarily, they should have an increased temporary count to facilitate this.

One way to accomplish the shift is recalculating all node temporary counts every so often. The temporary counts T_j at all nodes j are added up, divided by the number of nodes n , and each node given the quotient as its new temporary count.

A more refined way to accomplish the shift is to have each node maintain a running sum of how much temporary “work” W_j it has done so far, that is, each time a node is the first to perform a transaction temporarily, the node adds to its “work” sum W_j the amount of the transaction r_i . Then, instead of using a simple average for each node’s new temporary count, a weighted average is used.

Such summing of all T_j requires a “snapshot” of the system and suspending or freezing the temporary processors until they get their new counts. We are investigating ways to redistribute temporary counts that do not involve such a suspension.

We are also investigating ways to include in the system donations of resources. A donation is added to the permanent counts P_j of all nodes. It is desirable for a donation to be incorporated immediately into the temporary and permanent counts by the permanent processors rather than being added to the end of a possibly lengthy parent queue.

6 Conclusion

COPAR has been shown to be an effective system for the distributed handling of reservations over a slow or congested network. A flaw was discovered and fixed, preserving the performance of the system. Future enhancements to incorporate donations and shift temporary counts where needed are anticipated.

7 References

- [1] C. Innis, J. Crichlow, M. Hosein, S. Hartley. A Java System that combines Optimism and Pessimism in Updating Replicas, Proceedings of the Sixth IASTED International Conference on Software Engineering and Applications, Cambridge, Massachusetts, USA, November 4–6, 2002.

- [2] J. Crichlow, S. Hartley, M. Hosein, C. Innis. The COPAR Service: Combining Optimism and Pessimism in Accessing Replicas, Proceedings of the Third IASTED International Conference on Communications, Internet, and Information Technology, St. Thomas, US Virgin Islands, November 22–24, 2004, 558–563.
- [3] J. Crichlow, S. Hartley, M. Hosein, D. Ivins. Using an XML File to Test a Distributed Replica Accessing System, Proceedings of the Fourth IASTED International Conference on Communications, Internet, and Information Technology, St. Thomas, US Virgin Islands, November 29–December 1, 2006, 324–329.
- [4] J. Crichlow, S. Hartley, M. Hosein. Using COPAR to Facilitate Quick Distribution of Disaster Relief, Proceedings of the Ninth IASTED International Conference on Software Engineering and Applications , Orlando, Florida, USA, November 16–18, 2008, 100–105.
- [5] S. Beharry, M. Hosein, S. Hartley, J. Crichlow. Processing Transactions With Greater Accuracy and Availability in COPAR, Proceedings of the 21st IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, Massachusetts, USA, November 2–4, 2009, 21–28.

A Novel Cloud Computing Data Fragmentation Service Design for Distributed Systems

Ismail Hababeh

School of Computer Engineering and Information Technology, German-Jordanian University
Amman, Jordan

Abstract- *As many distributed database applications contain online information that change continuously and expand incrementally, comprehensive cloud Application Programming Interface API's are required to monitor and control the accuracy of the information and data proliferation. This cloud software is required to monitor and control the accuracy of the information and data proliferation. This software can be viewed as integrated cloud computing services; data fragmentation, clustering network sites, and fragments Allocation that support transactional database applications. In this paper, we describe our data Fragmentation as a Service (FaaS) in construction of a cloud computing software system. Specifically, we design a novel data fragmentation as a service to facilitate enormous data processing, and introduce some functioning enhancement on data distribution to improve the cloud system performance. This research presents our attempt to implement data fragmentation service in a cloud computing system, with large scale data mining as targeted application.*

Keywords: SaaS, FaaS, CaaS, AaaS, DFA, API.

1 Introduction

Cloud computing is web based system development in which huge scalable computing services are provided to users over the Internet. The cloud computing system includes web communications, Software as a Service (SaaS), up-and-coming tools, and has involved extra attention from researchers in different technology areas.

Many cloud computing providers have their data centers spread worldwide to maintain data availability which is typically achieved by replication processes. Amazon's cloud simple storage service [1] replicates data across different geographical regions so that data and applications can continue even in the face of failures of their location. This is likely to be help in running applications on data warehouses, but not transactional data management systems [2].

Yahoo [3] and Amazon [4] both implement data replication through PNUTS and SimpleDB cloud data services over distributed network sites. They designed to run analytical applications on data warehouses, but not for transactional data applications. Similarly, Google [5] implements a replicated database, but does not offer a complete relational Application Programming Interface and weakens the data atomicity. The cloud API is written as series of XML-based messages, and executed on the cloud servers to utilize remote web-based applications and reduce the number of calls between the client and the distributed servers [6].

Microsoft SQL Server [7] cloud data service is implemented over distributed network sites. However, as it doesn't apply commit protocols, the distributed system presents lack of data consistency. Researchers in [8] designed the H-Store project to minimize the number of transactions that access data from multiple locations. However, the project still in the theoretical phase, and its feasibility on a real world distributed database systems has not verified.

In distributed relational database systems, the transactions on the applications are usually subsets of relations (fragments), so using these fragments and distributing them over the network sites increases the system throughput by means of parallel execution. Therefore, an efficient cloud API fragmentation web service is presented to access and manage data relationships, and enhance both the speed and simplicity of the distributed database functionality. This web service is used to retrieve raw data from the cloud data centers by external programs like Java applications. Moreover, it helps to reduce the cost of accessing data over distributed network sites and increases the distributed system performance through data allocation processes.

The remainder of the paper is organized as follows: related work is discussed in Section 2; Section 3 describes the data fragmentation architecture; data fragmentation design is presented in Section 4; Section 5 depicts the performance evaluation and experimental results; and finally Section 6 draws conclusion and outline future work.

2 Related Work

Various strategies have already partitioned data across distributed systems. There are approaches that determined three main partitioning categories; vertical, horizontal, and hybrid [9, 10]. Some have defined the vertical fragmentation as a process of generating data records fragments [11, 12, 13, 14]. Other researchers have addressed the necessity of horizontal fragmentation [15, 16, 17] which make the processes of data backup and restore much easier. A mixed or hybrid fragmentation; vertical fragmentation followed by a horizontal or vice versa, has been covered by few researches [12,18] due to the intractable nature of this type of fragmentation in relational distributed database systems.

The studies in [16] and [17] are by far the closest to our fragmentation method. The method in [16] considered each record as a fragment in the relation and large number of database fragments is generated, thus more communication costs are required fragments processing. In contrast, the approach in [17] used the whole relation as a fragment, not all records of the fragment have to be retrieved or updated, and a selectivity matrix that indicates the percentage of accessing a fragment by a transaction is considered. However, more redundant fragments are available, and the generated fragments are overlapped.

A key difference between our cloud API fragmentation method and the others is that: it presents the minimum number of disjoint fragments that could be generated for each relation according to the queries requirements. This fragmentation service is designed for a cloud computing systems in order to reduce the communication cost over the cloud sites and increase the distributed system throughput. Moreover, the generated disjoint fragments are allocated then into the cloud servers where it saves more communication costs.

3 Data Fragmentation Architecture

In a distributed relational database systems, the complete database is not a suitable data unit for distribution because it is too big, especially when considering information relevant for different cloud data centers. Therefore, it is appropriate to develop a web application programming interface service, specifically FaaS, that can extract the minimum number of disjoint data records which would be allocated to the cloud servers.

The architecture of FaaS is recognized by the domain knowledge and three main processes; eliminating data redundancy, defining transactions, and fragmenting data records. Figure 1 describes the Data Fragmentation Architecture (DFA) service that will be used for generating

data fragments, supporting the use of knowledge extraction, and helping to achieve the effective use of small fragments.

The domain knowledge in DFA describes and categorizes the essential and representative elements of the distributed database systems, specifically, for the databases fragmentation. The purpose of the DFA domain knowledge is to ensure that all data elements are available and consistent for database fragmentation process. In addition, it is used to prepare data elements that are valid from one transaction to another, from one application to another, and from one database to another in distributed database systems. The details of this DFA are described and illustrated in the following section.

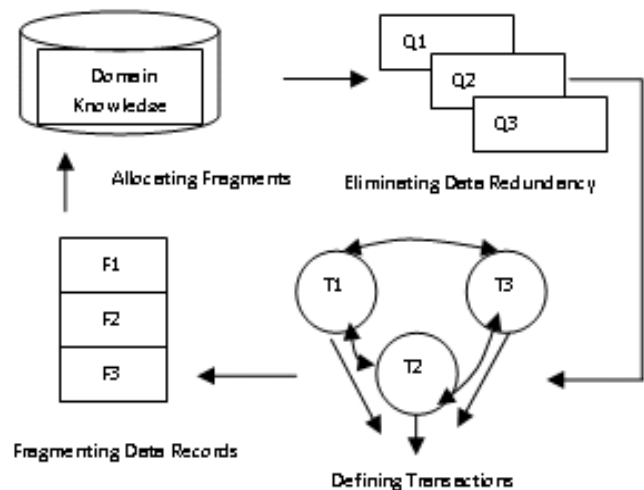


Figure 1. Data Fragmentation Architecture

4 Data Fragmentation Design

The requested data in DFA are identified by means of transactions triggered as queries, which determine the specific information that should be extracted from the cloud database servers. The transactions are executed and result in redundant data records as two or more different queries may require the same data records. The redundant data are eliminated and the remaining data records are then partitioned so as to generate the minimum number of fragments which are neither replicated nor intersected (disjoint).

4.1 Eliminating Data Redundancy

Different cloud API's are developed to get rid of data redundancy from the distributed cloud servers. The following algorithm is designed to prevent data replication, based on Primary Code Number PCN, from being entered into database application runs over cloud servers.

Eliminating Data Redundancy Algorithm:

-
- Step1: Build a map which stores the unique list of leads being inserted/updated, using Primary Code Number as a key.*
 - Step2: Check for any lead where the key is inserted/updated*
 - Step3: If the key is a replication of another lead in this group, Then do steps 4 and 5*
 - Step4: Create a single database query, using the lead map, to find all the leads in the database that have the same key*
 - Step 5: Issue a non validation message*
 - Else, do step 6*
 - Step 6: If the key is not replicated, Then*
 - Add this lead to the new lead map*
 - End If*
 - End If*
 - Step 7: End*
-

In this cloud API, all transactions are processed and the redundant data records are eliminated. Thus, database applications get more speed and so more efficient as it have only the required data records to be accessed, processed, and allocated to the distributed cloud servers.

4.2 Defining Database Transactions

The data records requested by the clients determine the specific information extracted from the database queries. Database queries are executed as transactions from the applications at the distributed database system sites. The results of the transactions are sets of data records that could be full intersected, partial intersected, or not intersected. The data set itself consists of complete records.

Site #	Transaction #	Transaction Definition
S1	T1	List of all employees
S2	T2	Accountant employees at site 2
S3	T3	Ranks of employees at site 3
S7	T4	List of employees of ranks <= R4
S7	T5	Sites at locations between L-11110 and L-11115
S8	T6	Employees who have basic salary > \$ 3500
S9	T7	Employees who have allowances >= \$ 500
S10	T8	Ranks where allowances <= \$ 450
S11	T9	List of engineers at site 11
S12	T10	List of sites where site# <= 7

Figure 2a. Defining Transactions

Figures 2a and 2b illustrates an example of defining and generating different sets of data records according to the definition of each transaction.

Data Set #	Site #	Transaction #	Relation #	Record(s) #
1	S1	T1	4	1,2,3,..., 20
2	S2	T2	5	1,5,6,7
3	S3	T3	2	1,2,3,4,5,6,7
4	S4	T4	3	1,2,3,..., 12
5	S5	T5	4	1,2,3,4
6	S6	T6	1	1,2,3,..., 20
7	S7	T7	1	3,18
8	S8	T8	5	2,3,4
9	S9	T9	2	2,3,4
10	S10	T10	3	1,2,3,..., 12

Figure 2b. Data Records Transactions

In this figure, there exist a full intersection between data sets (4,10) over relation 3, and a partial intersection between data sets (1,5), (3,9), and (6,7) over relations 4, 2, and 1 respectively. On the other hand, there is no intersection between the data sets over relation 5. Therefore, a cloud API fragmentation method is developed, partitioned the database records, and generated the minimum number of disjoint fragments which will be allocated to the distributed cloud servers. The details of this approach are illustrated in the following section.

4.3 Fragmenting Data Records

The fragmentation process starts looking for any two data records over the same relation having intersection records between them. From any two intersected data sets, three disjoint fragments will be generated; the intersection fragment which represents the common records in both sets, the fragment that represents the records in the first set but not in the second intersected set, and the fragment that represents the records in the second set but not in the first intersected set. Then, the intersected sets are deleted from the data sets list. This process is continued until no more intersections between the data sets still exist. The subsequent fragmentation algorithm describes the processes of generating disjoint fragments from the intersected data records for each relation in DDDBS.

Intersected Data Fragmentation Algorithm:

$k \leftarrow$ Number of the last fragment in the database (0 at the beginning)

Repeat for all relations in the database

Repeat for all data records S_i, S_j in each relation,

where $i \neq j$

If $S_i \cap S_j \neq \emptyset$

$k \leftarrow k + 1$

$F_k \leftarrow S_i \cap S_j$

$F_{k+1} \leftarrow S_i - F_k$

$F_{k+2} \leftarrow S_j - F_k$

Delete S_i, S_j

End if

Until all intersected data records in each relation have been processed

Until all relations in the database have been processed

Rename the final fragments sequentially

The data records over each relation that do not have intersection between them are renamed as fragments and considered for further fragmentation process. The following fragmentation algorithm expresses the non-intersected data records and adds them to the list of relation fragments.

Non-intersected Data Fragmentation Algorithm:

$k \leftarrow$ Number of the last fragment

Repeat for all relations in the database

$k \leftarrow k + 1$

$F_k \leftarrow R - \cup F_i$ (for all fragments F_i in relation R)

IF $F_k \neq \emptyset$ Then

Add F_k to the collected fragments of relation R

End if

Until all relations in the database have been processed

The same fragmentation process is applied for any two intersected fragments over the same relation. This re-fragmentation process will be continued until the intersection between data fragments is finished and the fragments are totally disjoint. Figure 3 shows an example of generating disjoint fragments from the data sets over relation 2 in a DDBS.

In this figure, the data sets 1,2 over relation 2 are sharing a common data records 2,3,4 which is considered to be redundant data. The fragmentation process isolates the shared data records from both data sets, and generates the following disjoint fragments; F1 which contains the shared data records 2,3,4, F2 that contains the records 1,5,6,7 which are in data set 1 but not in data set 2.

As the third fragment should be generated contains the same data records in fragment F1, it will not be created and the data sets 1,2 have to be deleted. The performance evaluation of the fragmentation method is presented in the following section.

Data Set #	Site #	Transaction #	Relation #	Record(s)#
1	S1	T1	2	1,2,3,4,5,6,7
2	S2	T2	2	2,3,4

Fragment #	Site #	Transaction #	Relation #	Record #
F1	S1,S2	T1,T2	2	2,3,4
F2	S1	T1	2	1,5,6,7

Data Set #	Site #	Transaction #	Relation #	Record(s)#
3	S3	T3	2	1,2,3,4,5,6,7
4	S4	T4	2	1,2,3,4,5,6,7

Fragment #	Site #	Transaction #	Relation #	Record #
F3	S3,S4	T3,T4	2	1,2,3,4,5,6,7

Fragment #	Site #	Transaction #	Relation #	Record #
F1	S1,S2	T1,T2	2	2,3,4
F3	S3,S4	T3,T4	2	1,2,3,4,5,6,7

Fragment #	Site #	Transaction #	Relation #	Record #
F5	S1,S2,S3,S4	T1,T2,T3,T4	2	2,3,4
F6	S3,S4	T3,T4	2	1,5,6,7

Figure 3. Data Sets Fragmentation

5 Performance Evaluation and Experimental Results

The database fragmentation performance evaluation is based on the computation resulted from dividing the reduced storage size for each relation by the relation queries' records size. The reduced storage size is computed as the difference between the size of the queries' records and the size of the generated fragments of each relation in DDBS.

The experimental results are demonstrated the usability and the efficiency of the cloud API fragmentation method. When this method is tested for 45 queries over 5 database relations that construct the whole database, 27 disjoint fragments are generated. Therefore, 18 data set records are omitted from the database which eliminates the data redundancy, saves more data storage, minimizes the transferred and processed data, and then increases the overall system performance.

The fragmentation methods in [16] and [17] are implemented in our fragmentation method. The performance results of [16] and [17] are compared with our cloud API fragmentation method and depicted in Figure 4.

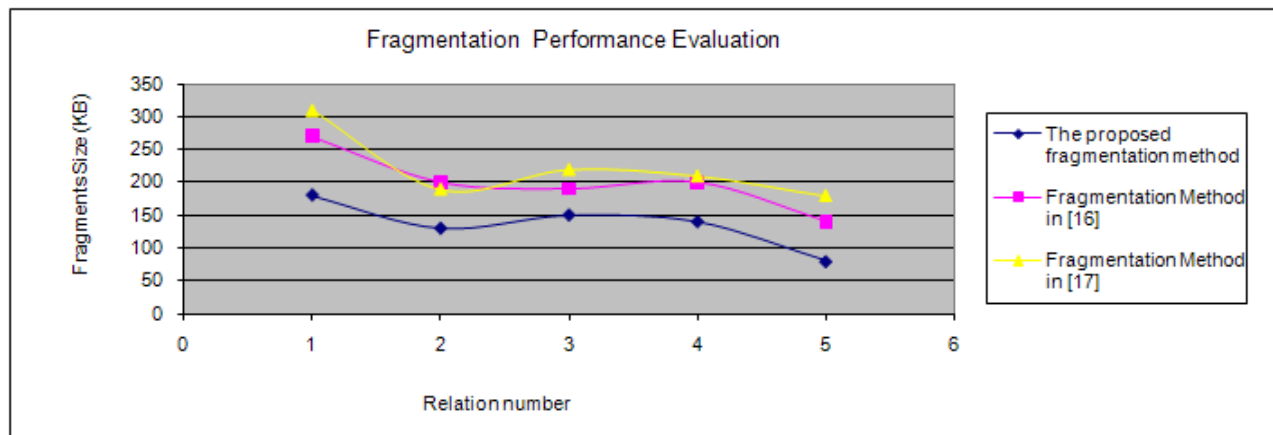


Figure 4. Fragmentation Performance Evaluation

It is shown in this figure that the performance accomplished by our approach outperforms the performance of the methods in comparison, and this will help in reducing the communication costs in data allocation phase.

6 Conclusions

Data fragmentation is one of the primary techniques used in partitioning and developing cloud computing services for distributed database systems. This research discussed the efficiency, usefulness, and the performance improvement achieved by the API fragmentation service in a cloud computing system. The experimental results emphasized the ability of this fragmentation method to minimize the data processed and transferred between the distributed database system network sites, reduce the storage size by eliminating data redundancy, and present significant performance improvements that increase distributed database network system throughput.

Moreover, this cloud API fragmentation approach realizes the optimal solution properties of data fragmentation in a distributed database system; the relation fragments include all relation records, the union of all relation fragments constructs the original relation, and the relation fragments are disjoint. In addition, it generates the minimum number of fragments for each relation according to the queries' requirements. This will reduce the communication cost over distributed network sites and increase the distributed system performance.

In a future work, the research will be focused on the resulted disjoint fragments as objects for distribution over the network sites. It should be distributed in such a way that satisfy the requirements of database queries and enhance the system performance. Therefore, developing web API services for a cloud computing distributed

systems, like Clustering distributed network sites as a Service (CaaS) and fragments Allocation as a service (AaaS), have an important impact on the transactional database applications, and improve the distributed systems throughput.

References

- [1] Amazon Simple Storage Service (Amazon S3) <http://aws.amazon.com/s3> [Accessed 29th April, 2011].
- [2] Daniel J. Abadi. Data Management in the Cloud: Limitations and Opportunities. Data Engineering, IEEE Computer Society. March 2009 Vol. 32 No. 1, pp 3-12.
- [3] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. Proceedings of VLDB, 2008.
- [4] Amazon Simple DB. <http://aws.amazon.com/simpledb/> [Accessed 19th February, 2011].
- [5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. Proceedings of OSDI, 2006.
- [6] A. Velte, T. Velte & R. Elsenpeter. Cloud Computing: A Practical Approach. McGraw-Hill. 2010.
- [7] Microsoft SQL Server for Cloud Servers. <http://www.rackspace.com/cloud/blog/2010/12/01/announcing-sql-server-licenses-for-cloud-servers>. [Accessed 7th March, 2011].
- [8] M. Stonebraker, S. R. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). VLDB, Vienna, Austria, 2007.

- [9] Ozsü, M. & Valduriez, P. Principles of Distributed Database Systems. 2nd ed. Englewood Cliffs NJ, Prentice-Hall. 1999.
- [10] Khalil, N., Eid, D. & Khair, M. Availability and Reliability Issues in Distributed Databases Using Optimal Horizontal Fragmentation. Trevor J. M. Bench-Capon; Giovanni Soda & A. Min Tjoa, ed., 'DEXA' 99, LNCS 1677, Springer, pp. 771-780. 1999.
- [11] Son, J. & Kim, M. An Adaptable Vertical Partitioning Method in Distributed Systems. The Journal of Systems and Software. 73(3), 2004, pp. 551 – 561.
- [12] Agrawal, S., Narasayya, V. & Yang, B. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. SIGMOD 2004, Paris, France, ACM 2004, pp. 359-370.
- [13] Tamhankar, A. & Ram, S. Database Fragmentation and Allocation: An Integrated Methodology and Case Study. IEEE Transactions on Systems, Man, and Cybernetics-Part A. Systems and Humans. 28(3), 1998, pp. 288 – 305.
- [14] Lim, S. & Ng, Y. Vertical Fragmentation and Allocation in Distributed Deductive Database Systems. The Journal of Information Systems. 22(1), 1997, pp. 1-24.
- [15] Costa, R. & Lifschitz, S. Database Allocation Strategies for Parallel BLAST Evaluation on Clusters. Distributed and Parallel Databases. 13, 2003, pp. 99-127.
- [16] Ma, H., Scchewe, K. & Wang, Q. Distribution design for higher-order data models, Data and Knowledge Engineering. 60, 2007, pp. 400-434.
- [17] Huang, Y. & Chen, J. Fragment Allocation in Distributed Database Design. Journal of Information Science and Engineering. 17, 2001, pp. 491-506.
- [18] Navathe, S., Karlapalem, K. & Minyoung, R. A mixed fragmentation methodology for initial distributed database design. Journal of Computer and Software Engineering. 3(4), 1995, pp.395-425.

Parallelizing Tompa's Exact Algorithm for Finding Short Motifs in DNA

Christopher T. Mitchell¹, Jonathan Grochowski¹, Julian H. Dale¹, Nicolas B. Wilson¹, and Jens Mache¹

¹Department of Mathematics & Computer Science, Lewis & Clark College, Portland, Oregon, USA

Abstract—*Motif finding, the search for regulatory sequences in DNA, is a computationally expensive challenge in bioinformatics. This paper presents a pleasantly parallel version of Tompa's exact method for finding short motifs. We use a distributed-memory computer cluster and MPI to run our parallel algorithm and collect data. We vary motif length and allowed substitutions. Results indicate good speedup and scalability.*

Keywords: cluster, parallel algorithm, bioinformatics, motif finding, performance evaluation, MPI

1. Introduction

A motif is a short sequence of DNA that has a specific function and appears multiple times throughout a genome. A motif could be many things, including a transcription factor binding site, or a ribosome binding site. Motifs are of interest to biologists because they often play important roles in the regulation of gene expression.

Finding a motif amongst a set of DNA sequences is not a trivial task. The motif may not appear in every single sequence, and instances of the motif may not be identical due to substitutions, insertions, and deletions. To find this motif, one not only needs to accommodate for inexact matches, but one must also devise a way to filter the true biological motifs from patterns that randomly occur within the sequences.

Our initial survey of DNA motif finding algorithms showed that the set of approaches to this problem is very diverse. Within this set, there are two general approaches to the problem [1]. The first approach uses a word-based algorithm that analyzes a string of nucleotides and counts and compares the frequency of specific k -mers (contiguous substrings of length k). Word-based algorithms rely on exhaustive enumeration and can guarantee an optimal result. The second approach involves using probabilistic models where the parameters are based on some form of statistical inference (maximum-likelihood, Bayesian) or weight matrix.

This paper will focus on an exhaustive word-based algorithm designed by Tompa [2]. Our goal was to decrease the run-time of this algorithm by parallelizing its execution. This paper will present our method of parallelization, performance results, and suggestions for continued work.

2. Parallelizing Tompa's algorithm

2.1 Tompa's word-based algorithm

Tompa developed a word-based algorithm that takes a set of DNA sequences and a k -mer length as its input, and outputs z -scores for all motifs (words) of length k . The algorithm was designed to overcome two weaknesses that he identified in more naive word-based motif finding approaches [2]. These naive algorithms (that simply count k -mer frequency or measure k -mer entropy) are vulnerable to improperly scoring motifs when either the background nucleotide distribution is not uniform or when pairs of motifs occur in largely different number of input sequences.

Tompa's approach was to score each k -mer with a z -score constructed from the observed and expected number of input sequences that have an occurrence of the given k -mer. The z -score for some k -mer s is given by

$$M_s = \frac{N_s - Np_s}{Np_s(1 - p_s)}, \quad (1)$$

where N is the number of input sequences, N_s is the number of input sequences that contain an occurrence of s , and p_s is the probability of observing an occurrence of some s in a random sequence. The idea of the z -score and the technique for calculating p_s was the crux of Tompa's work, but not necessarily the most computationally expensive. It is worthwhile to impress that calculating N_s involves examining every input sequence in turn with respect to k -mer s .

2.2 Identifying opportunities for parallelization

To help us determine which portions of the algorithm would benefit most from parallelization, we profiled our sequential implementation of the algorithm (pseudocode in Figure 1). For our profiling, we ran our program with parameters that mimicked those defined by Tompa in the motivating computational problem in Section 1.1 of his paper: 4000 input sequences, each 20 nucleotides long, searching for 5-mer motifs [2].

The profiling revealed that the 64% of CPU time was spent in the routine that checks to see if an input sequence contains (or "matches") an occurrence of a k -mer (line 5). There were 4,096,000 calls to that matching function — more calls than were made to any other function. This number, while

```

1 for kmer in kmer_set:
2     p_s = ...
3     N_s = 0
4     for sequence in input_sequences:
5         if kmer occurs in sequence:
6             N_s += 1
7     M[kmer] = calc_z_score(N_s, p_s)
8
9 return top_kmers(M)

```

Fig. 1: Pseudocode for serial calculation of the z -scores

surprising, can be understood by realizing that for each k -mer, we check for matches against all sequences:

$$4^5 \text{ } k\text{-mers} \times 4000 \text{ sequences} = 4,096,000 \text{ checks.} \quad (2)$$

Even though checking for a match is fast, by virtue of there being so many checks, counting N_s takes more time than any other step in the algorithm. The functions that calculate p_s (line 7), for example, are only called once per k -mer ($4^5 = 1024$ many times in this case) regardless of the number of input sequences.

The simplest way to break up the work of calculating N_s is to split the work up at the level of calculating M_s , since M_s is dependent on both N_s and p_s . Said differently, parallelizing the outer loop (line 1) of the algorithm will yield the easiest and most immediate gain. Only 0.3% of the program's execution time was spent outside of the loop.

2.3 Method of parallelization

Since the calculations in the step that we identified could be easily partitioned into separate (and independent) jobs that do not need communication, the task is well suited to run on a distributed computer like a Beowulf cluster with MPI. We assembled our own Beowulf cluster, called BeoPup, composed of 18 single-core AMD 64-bit Athlon processor nodes connected by a TCP/IP Ethernet switch. For parallelizing the motif finding algorithm on our BeoPup cluster, MPI was a natural choice because it is designed to enable parallel computing for interconnected machines that do not share memory, like our cluster. The Beowulf design of our cluster was well suited for the low communication requirements of our parallelization of Tompa's algorithm. To use MPI with our Python implementation, we settled on the mpi4py module [3] because of its active development.

To split the work of calculating the z -scores among multiple computers, we wrote a SPMD (single program, multiple data) type program that would calculate the z -scores for disjoint subsets of all of the k -mers, where the subset is dependent on which node in the cluster the program is running on. In MPI terminology, we used a "gather" at the end of the computations so that the z -scores each node calculated are sent to a master node (given rank zero in our

example) that sorts and outputs the best scoring k -mers. The source code for this parallel implementation has been made available online [4].

3. Results

We ran our program with the same parameters given earlier on 1, 2, 4, 8, and 16 nodes. We recorded the shortest of five run-times for each condition in Table 1. Running the program on 16 nodes, for example, yielded a nearly 15 times speedup over the time it takes to run the program on just a single node. The efficiency for this 16 node case is given by the speedup divided by the number of nodes: $\frac{14.996}{16} = 0.937$. An efficiency of 1 would indicate a perfectly linear speedup, where the parallelized run-time is equal to the single-node run-time divided by the number of available nodes. The slightly decreasing efficiency can be explained by communication overheads and the redundant computation of the background nucleotide Markov model (used in finding p_s) that occurs on all nodes. A near perfectly linearly speedup of our algorithm holds until the number of available processors nears the granularity of the parallelization, which is given by the number of k -mers.

Table 1: The run-time of our parallelization of Tompa's algorithm decreases almost linearly with respect to the number of compute nodes.

Nodes	Speedup	Efficiency	Time (seconds)
1	1.000	1.000	37.227
2	1.971	0.985	16.873
4	3.894	0.974	8.539
8	7.706	0.963	4.315
16	14.996	0.937	2.217

Figure 3 shows run-time increasing exponentially when we increase the length of the motifs that we are searching for. Because of the design of our parallelization, increasing the problem size in this way will only increase the efficiency when running on many nodes. This is because the proportion of time spent calculating z -scores over the time spent in non-parallel code increases as k increases.

4. Discussion and Future Work

We have shown that the application of parallel techniques can greatly increase the speed of Tompa's motif-finding algorithm. Others have demonstrated success parallelizing widely-used motif finding algorithms; MEME is one such example that uses clusters of GPUs [5]. It reasons that other motif-finding algorithms could benefit from the application of parallel computing techniques and that the resultant speedup could make running precise (but expensive) algorithms a more practical prospect.

After the completion of the parallelization of Tompa's algorithm, we reached out to members of the bioinformatics community. So far, the improvements were met with a

```

1
2 for kmer in kmer_set:
3     p_s = ...
4     N_s = 0
5     for sequence in input_sequences:
6         if kmer occurs in sequence:
7             N_s += 1
8     M[kmer] = calc_z_score(N_s, p_s)
9
10
11
12
13 return top_kmers(M)

```

Serial version

```

kmer_set = get_kmers_for_rank(my_rank)
for kmer in kmer_set:
    p_s = ...
    N_s = 0
    for sequence in input_sequences:
        if kmer occurs in sequence:
            N_s += 1
    M[kmer] = calc_z_score(N_s, p_s)

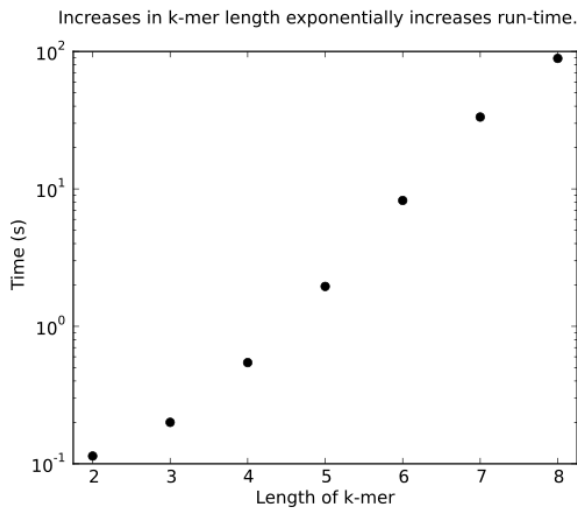
M = gather_to_rank_0(M)

if my_rank is 0:
    return top_kmers(M)

```

Parallel version

Fig. 2: Pseudocode of parallelization

Fig. 3: Increases in k -mer length exponentially increase run-time.

variety of feedback as well as some constructive criticism to help form a plan for the future of our work with Tompa's algorithm. An insightful comment was provided by Dr. Jonathan Visick, a professor of Microbiology and Genetics at North Central College,

With hundreds of new bacterial whole-genome sequences being completed each year, problems like identifying ribosome-binding sites are not going away: the sequences are not the same for different species, necessarily. Plus, an algorithm for finding a ribosome-binding site presumably would also be applicable to finding promoters in prokaryotes, transcription-factor binding sites in eukaryotes and various other sequence features. [6]

The next step for this algorithm probably entails a more col-

laborative effort from within the bioinformatics community. The feedback indicates that the importance of motif-finding is not dwindling and that their utility is greater now than ever before. Since Tompa's algorithm does not make any biological assumptions, it can be adapted to address some of the challenges mentioned by Dr. Visick.

Tompa's algorithm involves creating a distinct deterministic finite automaton (DFA) for each k -mer. This DFA is used to match sequences that have an occurrence (recall that an occurrence allows substitutions) of the related k -mer, and also to calculate p_s . As a possible extension to his work, Tompa suggests finding a way to accommodate longer k -mers and more substitutions. This is because the DFA creation algorithm that we implemented, while computationally fast for simple cases that allow only one substitution, becomes slow enough when it is modified to allow for more substitutions that the program's total run-time is markedly increased (Figure 4).

Profiling our code showed that DFA creation (instead of matching) takes up the majority of the run-time when we allow for three substitutions. Although Tompa suggests core modifications to his algorithm to allow for more substitutions, simply adding more compute nodes to decrease the number of k -mers that each node must inspect might be seen as a possible solution to the problem. For example, the many-thousand cored super computers ranked in the TOP500 list could easily tackle a problem with limited substitutions [7].

5. Conclusions

Motif finding is a computationally expensive task. This paper presented a parallel version of Tompa's exact method for finding short motifs [2]. Using a Beowulf cluster, we showed that our parallel version of the algorithm was able to significantly speed up the search for motifs. For example, when searching for a motif using 16 nodes, we achieved

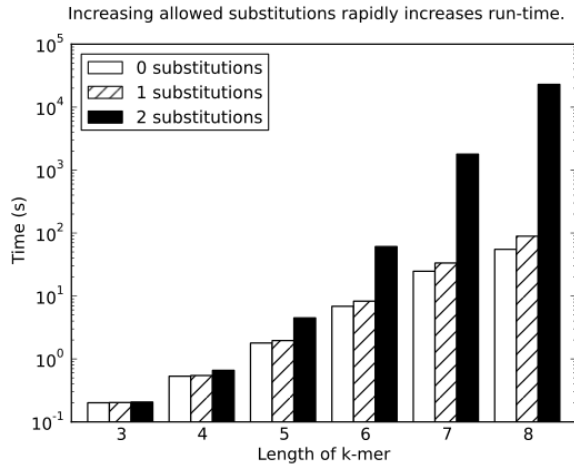


Fig. 4: For large k -mers, increasing the number of allowed substitutions significantly increase run-time.

a speedup of almost 15 times. Parallel motif-finding algorithms can enable searching for more complex motifs in reasonable amounts of time.

6. Acknowledgements

We would like to thank Dr. Jonathan Visick, Dr. Deborah Lycan and Dr. Adam A. Smith for discussions about biology and bioinformatics. This material is based upon work supported by the John S. Rogers Science Research Program at Lewis & Clark College and by the James F. and Marion L. Miller Foundation.

References

- [1] M. Das and H.-K. Dai, "A survey of dna motif finding algorithms," *BMC Bioinformatics*, vol. 8, no. Suppl 7, p. S21, 2007. [Online]. Available: <http://www.biomedcentral.com/1471-2105/8/S7/S21>
- [2] M. Tompa, "An exact method for finding short motifs in sequences, with application to the ribosome binding site problem," in *Intelligent Systems in Molecular Biology*, 1999, pp. 262–271.
- [3] "MPI for Python," <http://mpi4py.scipy.org/>.
- [4] "Code : motif-finder," <https://code.launchpad.net/motif-finder>, 2011.
- [5] Y. Liu, B. Schmidt, and D. L. Maskell, "An ultrafast scalable many-core motif discovery algorithm for multiple gpus," in *2011 IEEE International Parallel & Distributed Processing Symposium*.
- [6] J. Visick, personal communication, 2011.
- [7] "TOP500 List - November 2010," <http://top500.org/list/2010/11/100>, 2010.

RNS: Remote Node Selection for HPC Clusters

Seyedeh Leili Mirtaheri, Ehsan Mousavi Khaneghah, Siavash Ghasvand, Mohammad Norouzi Arab, Ashkan Shirpour and Mohsen Sharifi

School of Computer Engineering
Iran University of Science and Technology
Tehran, Iran
{msharifi, mirtaheri, emousavi}@iust.ac.ir
{noroozi, ghasvand, a_shirpour}@comp.iust.ac.ir

Abstract

Scalability of distributed high performance computing clusters in different administrative domains is critically dependent on the deployment of a proper resource discovery mechanism that can discover resources residing in different wide administrative domains. In this paper, we present a new method for remote cluster node selection called RNS that such a required resource discovery mechanism may use. We show analytically that RNS is superior to existing resource discovery mechanisms in response time and utilization of free resources.

1. Introduction

The performance of a traditional high performance computing (HPC) cluster comprising of a number of homogenous computers interconnected by a dedicated local area network is limited by the amount of available resources in the locality of the cluster. Better said, traditional HPC clusters are limited in both performance and scalability. Some clusters [1] just ignore requests for resources higher than available in the cluster, while some others [2] keep some limited number of resources out of the cluster as reserved and use it in case there are requests for higher number of resources. Both approaches fail to remove performance and scalability limitations.

One solution to improve the performance and scalability of HPC clusters is to allow them to utilize out-of-cluster accessible and available distributed resources whose types may be different from the inside-of-the cluster resources i.e. are heterogenous.

Such distributed clusters as any other distributed system promise higher performance and scalability. They however need a proper resource discovery mechanism that can discover heterogeneous resources residing in different wide administrative domains. A resource discovery mechanism is proper if it can satisfy as much as possible the dynamic resource requirements of every local process transparently irrespective of whether resources are local or remote.

The main challenge on the way of a proper resource discovery mechanism is that resources are heterogeneous and are dynamically added or crossed out from the set of available resources of the overall distributed system. Processes request for resources dynamically too implying that no prior knowledge exists on resource requirements of processes.

In this paper, we present a new method for remote cluster node selection called RNS that a proper resource discovery mechanism may use.

RNS does not require any agents on remote machines.

We have organized the rest of paper as follows. Section 2 presents a brief background on resource discovery and its operations. Section 3 reports notable related works. Section 4 presents and comparatively evaluates our proposed RNS method. Section 5 concludes the paper.

2. Background and Definitions

The duty of a resource discovery service in distributed systems is to find suitable resources for current requests generated in the system. Such a service is dependent on the daemons (agents) already configured on the member machines of the system [3, 4]. These daemons look after local resources so that they can provide accurate replies to the requests made by other daemons within the system.

The resource discovery approaches use different structures to support daemons to communicate. Three well-known structures are centralized, decentralized, and distributed [5, 6, 7, 8, 9]. However, regardless of the structure based on which a resource discovery service's daemons are formed, there is protocol according to which these daemons communicate. Therefore, it can be stated that the most important matter in communication of daemons is the common language they use [5]. This limits different resource discovery services' daemons in making communications with each other. The domain of a resource discovery service is thus defined by its daemons and the machines running daemons.

A resource discovery service performs well if resource requests can be satisfied using resources of the member machines of a cluster system. But what happens when all these resources are in use? How the resource discovery service can handle new requests? To answer these questions, we need to determine more accurately the area of work of the resource discovery service. To do

this, we introduce two areas. The first area contains member machines of a cluster system and the second one includes accessible systems that are not member of the cluster system.

In the first area, everything is almost preconfigured and foreseen. All system structures are determined and configured to satisfy the requirements of the cluster system. As a result, daemons are installed on every machine. These systems usually are homogenous or at least have little heterogeneity.

The second area discusses a completely unknown and heterogeneous world in which any resource of any type can exist. These resources are managed by different operating systems. Also, the member machines of such a system are not preconfigured and subsequently their required daemons are not installed on the machines. As a result, a uniform approach cannot be used for interacting with all of the machines. However, as mentioned earlier, since a resource discovery service needs to have some daemon preconfigured on all the machines, it can only work in the first area. To overcome the resource shortage inside the cluster, we need to use of out-of-cluster resources.

Because a resource discovery service is unable to work beyond the borders of a cluster system [10], a new service is required to handle requests using out-of-cluster resources. To implement resource discovery mechanisms on out-of-cluster resources, RNS must register those resources as members and then apply those mechanisms. RNS is only responsible for finding potential out-of-cluster resources and after selecting them some other mechanisms must be activated in order for the resources to be configured according to the requirements of the cluster system.

3. Related Work

There are few research works on the use of out-of-cluster resources. Some [2] use reserved

resources and some [1] ignore out-of-capacity requests. Only some researches like Mosix-2 propose instant joining of two cluster systems to each other to share their resources. The MOSIX Reach The Clouds Technology (MRC in short) determines how two MOSIX clusters can join to share their resources [11]. After joining, any node in the cluster can manage the cluster and dispatch its jobs. Therefore, the mentioned challenge is solved by adding available resources in other clusters. However, as stated before, some pre-configurations must be performed and both clusters must work under the same administrative domain.

Although a similar work on cluster systems (respecting the steps and their order) are not researched in depth, many works have been conducted on each single step.

Finding neighboring machines are so important that an IP protocol packet is devoted to it [12]. Two mechanisms for recognizing neighboring machines are introduced in [13] and [14]. These mechanisms work in P2P networks. On the other hand, many works have been conducted on recognizing resources of remote machines without using agents and many industrial tools exist in this field [15]. However, all of these tools use standards accepted and used by the world community. Among these tools we can refer to IBM Tivoli Monitoring [16] as two commercial tools and HP SiteScope [17] and Zabbix [18] and Nagios [19] as two open source tools.

4. The Proposed RNS Method

RNS has 4 phases that can be executed independently each. These phases include: request reception indicating the need for a resource, finding neighboring out-of-cluster machines, communicating with the discovered machines in order to fetch out their resource information, and finally selecting one of the

discovered machines regarding the requirements specified in the received message.

In RNS, first a message containing some information about the required resource is sent to one of the daemons of RNS located on the edge nodes of the cluster; an edge node is a node that is connected to the outside world. The receiving daemon analyzes the message and if relevant information exists it returns the information of the machine for joining it to the cluster. On the other hand, if there is no relevant information, the scanning process finds other neighboring out-of-cluster machines. In the next phase, communication with these machines starts. This phase is subdivided into two smaller phases. In the first sub-phase, the operating system of the remote machine is determined and in the second sub-phase the appropriate method for communication based on its operating system is chosen. In the final phase, the most suitable remote machine is chosen so that membership operations are performed. Also, any relevant information is cached so that further decisions are made faster.

RNS can act in a pro-active or passive manner. The previous scenario used the passive manner. In pro-active mode, before RNS daemons are sent messages, the scanning phase starts to handle requests faster.

The first phase of RNS is the only phase directly involved with the resource discovery mechanism. Therefore, it is better for the RNS service to have a well-defined interface so that it can communicate with as many resource discovery mechanisms as possible. In the second phase, it can make use of many mechanisms available in this field. In the third phase, using an alternating structure, this service brings the ability to communicate with many operating systems. Based on the operating system recognized at this phase, many tools and protocols like SNMP, SSH, DCOM, SLP and WMI can be utilized to

fetch information on remote machine's resources. Figure 1 depicts the relation between these 4 phases.

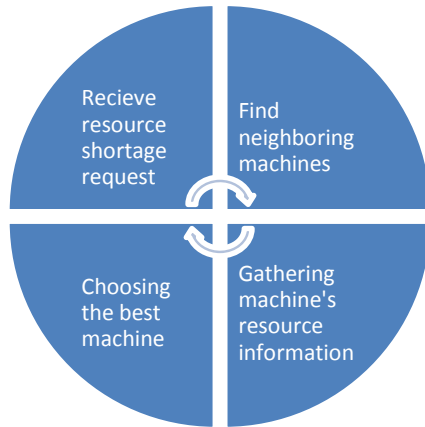


Figure 1: The relation between different phases involved in the RNS method

The pseudocode of RNS is presented below.

```

RNS Mechanism
/* A new request received */
Analyze_The_Request()
IF(Results_From_Previous_Searches_Are_Enough)
THEN
    /* RNS already has sufficient information and
    there is no need to search again */
    Compare_Available_Resources_And_Choose_The_Best()
ELSE
    /* Do discovery phases one by one */
    Scan_Neighborhood()
    Detect_Neighbors_OS()
    Gather_Neighbors_Resource_Info()
    /* choose the best result */
Compare_Results_And_Choose_The_Best()
END IF
/* the selected node is introduced to cluster for further
configurations */
Introduce_The_Selected_Machine_To_Cluster()
IF (Status==Passive) THEN
    // Daemon goes to standby mode until new request
    arrives
    Exit_And_Wait_For_New_Requests()
ELSE IF (Status==ProActive) THEN
    // Daemon returns to its previous state and
    continues scanning any reachable neighbor

Exit_And_Search_For_New_External_Resources()
END IF

```

5. Evaluation

The first method is responsible until a resource within the cluster is available. After all resources become unavailable, new requests are ignored. The second method differs from the previous method in the case all resources within a cluster are unavailable. At this situation it makes use of the reserved resources already preconfigured for such a situation. The third method or RNS equally behaves about the requests that can be answered using inter cluster resources. Requests that cannot be handled using inter cluster resources are sent to the RNS service so that some out-of-cluster resource are found for them based on the mechanisms already described.

One of the main parameters for determining the ability of a cluster system is its response time to requests. Therefore, we have chosen this parameter to compare our method with notable existing methods.

Figure 2 shows the response time regarding the number of requests generated. As depicted, the resource reservation method and RNS behave the same as the first method in the first interval (up to R1). Continuing, the first method ignores any more requests. Therefore, response time of this method goes to infinity as more requests arrive. However, the resource reservation uses reserved resources and can handle coming requests. The RNS method must find and add resources to the cluster from outside the cluster and it'll take much longer time to complete. Therefore, during the second interval, RNS is slower than the resource reservation method. By the beginning of the third interval all of the resources of the cluster and reserved resources are no longer available. In this situation, the first method still ignores requests and the second method has no available resources and must wait until some resource within the cluster itself or from the set of reserved resources become available. Therefore, its response time increases as more

and more requests are generated in the system. However, since RNS does not limit itself to a preconfigured set of resources, it never faces the problem of not having enough resources to handle requests. Although RNS shows lower performance during the second interval, it can compensate the lower performance in the third interval and yield a high performance overall.

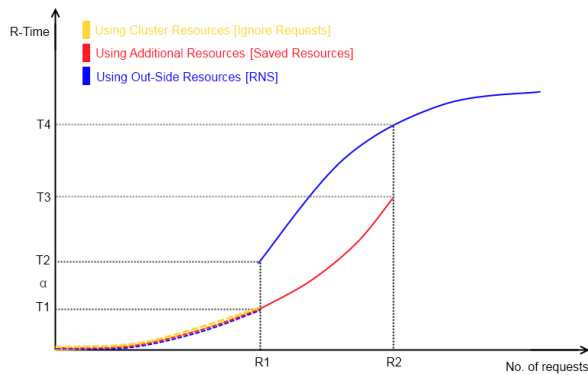


Figure 2: Comparison between the three methods in response time.

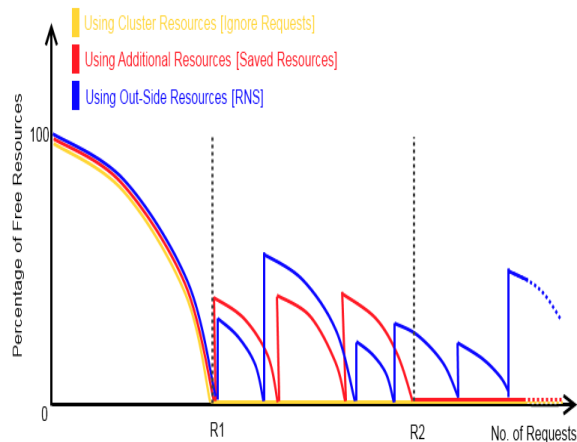


Figure 3: Comparison between the three methods in the use of free resources

At first, all of the resources within the cluster are free. As more requests are generated in the system, the number of available resources decreases. The three methods show the same behavior until there is no resource available

inside the cluster system. After all the internal resources become busy, the first method no longer responds and therefore there are no free resources. In this situation, the second method makes use of a set of reserved resources. The oscillations shown in this method's curve represent that the number of resources joining and leaving the cluster changes dynamically. Furthermore, as the number of requests exceeds the capacity of the resources of the cluster and the reserved set, no resources will be available to this method any longer. RNS is not limited to a preconfigured set of resources, therefore after all resources within the cluster become busy; it adds new resources from outside of the cluster. Therefore, it keeps oscillating and never stops responding requests.

6. Conclusion

In this paper we showed that our proposed remote cluster node selection called RNS responds well to cluster requests when cluster went out of resources. We also showed that we can prevent system failures in accomplishing requests and also prevent resources from being overloaded. We showed that RNS allows the use of external resources in the cluster system without doing any pre-reservation with high response time. Compared to other resource discovery methods, RNS has higher dynamicity, can interact with different operating systems, and does not need any agents on remote machines with which to communicate. These attributes has allowed the use of all out-of-cluster resources while keeping their utilization high. On the other hand, since no agent is used on remote machines and also its ability to interact with different operating systems, scalability is highly provided by RNS.

References

- [1] E. Pournaras, G. Exarchakos, and N. Antonopoulos, "Load-driven neighbourhood reconfiguration of Gnutella

- overlay," *Computer Communications*, vol. 31, no. 13, Aug. 2008.
- [2] D. Wischik, M. Handley, and M. B. Braun, "The resource pooling principle," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 5, Oct. 2008.
- [3] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat, "Design and implementation trade-offs for wide-area resource discovery," *ACM Transactions on Internet Technology*, vol. 8, no. 4, pp. 113-124, Sep. 2008.
- [4] M. L. Massie, B. N. Chun, and D. E. Culler, "The Ganglia Distributed Monitoring System: Design, Implementation And Experience," *Parallel Computing*, vol. 30, no. 7, pp. 817-840, Jul. 2004.
- [5] R. Raman, M. Livny, M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing", in Proceedings of HPDC, pp.140-140, 1998.
- [6] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat, "Scalable Wide-Area Resource Discovery," EECS Department, University of California, Berkeley, California, United States of America, Technical Report UCB/CSD-04-1334, 2004.
- [7] C. Mastroianni, D. Talia, and O. Verta, "A super-peer model for resource discovery services in large-scale grids", *Future Generation Computer Systems*, pp.1235-1248, 2005.
- [8] S. Ding, J. Yuan, J. Ju, and L. Hu, "A heuristic algorithm for agent-based grid resource discovery", *Intl. Conf. on e-Technology, e-Commerce and e-Service*, Hong Kong, pp.222-225, 2005.
- [9] D. Talia, P. Trunfio, and J. Zeng, "Peer-to-peer models for resource discovery in large-scale grids: a scalable architecture", *High Performance Computing for Computational Science – VECPAR 2006*, pp.66-78, 2007.
- [10] I.T. Foster, "Globus Toolkit Version 4: Software for Service-Oriented Systems", presented at J. Comput. Sci. Technol., pp.513-520, 2006.
- [11] A. Barak. (2011, May) MOSIX:Cluster and multi-cluster management. [Online]. <http://knol.google.com/k/amnon-barak/mosix/qibu8ltfp5fh/5>
- [12] T. Narten, E. Nordmark, W. Simpson, and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)," Network Working Group RFC 4861, 2007.
- [13] C. Mastroianni, D. Talia, and O. Verta, "A P2P Approach for Membership Management and Resource Discovery in Grids," in *Proceedings of the International Conference on Information Technology: Coding and Computing*, Washington, DC, USA, 2005.
- [14] P. Karwaczyński, D. Konieczny, J. Moćnik, and M. Novak, "Dual proximity neighbour selection method for peer-to-peer-based discovery service," in *Proceedings of the 2007 ACM symposium on Applied computing*, New York, NY, USA, 2007.
- [15] M. Sharifi, S. L. Mirtaheri, E. Mousavi Khaneghah, A *Dynamic Framework for Integrated Management of All Types of Resources in P2P Systems*, The Journal of Supercomputing, 52(2), 149-170, 2010.
- [16] T. Bhe, K. Inayama, C. Lister, M. Parlione, and M. Vesich, *Ibm tivoli monitoring version 5.1.1 creating resource models and providers*. Riverton, NJ, USA: IBM Corp, 2003.
- [17] H. Team, "Going beyond simple monitoring with HP SiteScope," Hewlett-Packard Company White Paper, 2010.
- [18] R. Olups, *Zabbix 1.8 Network Monitoring*. USA: Packt Publishing, 2010.
- [19] W. Barth, *Nagios: System and Network Monitoring*. San Francisco, CA, USA: No Starch Press, 2008.

Multimerge

Fernando Belmiro do Couto¹ and Fabio Silva do Couto²

¹Divisao de Ti, BBDTVM, Rio de Janeiro, RJ, Brasil

²UFRJ, Rio de Janeiro, RJ, Brasil

Abstract - *MultiMerge* is an algorithm that maps the ordered subsets of a mass of data, ordering them with Merge routines and using parallel processing. Most sort algorithms in use nowadays treat the mass of data without previously analyzing its distribution, no matter if dealing with full or partially ordered. According to the methodology applied in this algorithm the whole mass of data is in the worst case ordered each two elements, excepting its last element. According to the same method we prove that at random distributions, are statistically distributed among subsets of 2 + 3 elements. This algorithm tends to use the multiprocessing capability of current computers and is able to adapt itself to increasingly coprocessors quantity. The algorithm first scans a sequence of N elements, comparing each element with the next, verifying if they are ordered according to a previously established criterion and gathering them on positive or negative value subsets if they obey or not that criterion respectively. The major algorithm routine order two subsets using two threads simultaneously. First thread will merge from beginning until the half of subsets sum, comparing the smallest group elements. Second thread will merge from the end until the other half of the subsets sum, comparing the greatest group elements.

1. Introduction

Merging is faster than sorting. We can increase the speed using multithreads and if we have large ordered lists. Merge, multithreads and ordered lists are the focus of this algorithm. This is not a mergesort algorithm variant, since mergesort always divide the data the same way. The algorithm with performance comparable to this one is the algorithm proposed by MIT lecture 6172 "Analysis of multithreaded algorithms". That mergesort algorithm loses in average performance because in the beginning divide data always the every two elements and at the final threads because uses one thread to merge two lists. Multimerge main routine uses two threads to merge two lists. There is order in chaos it depends the way we look, in the first task of the algorithm we search for order.

2. Mapping Data

The algorithm first scans a sequence of N elements, comparing each element with the next, verifying if they are ordered according to a previously established criterion and gathering them on positive or negative value groups if they obey or not that criterion respectively. At the end of this

process, if the distribution is ordered, we have only one index which get a positive N value. On the other hand, if the distribution is reversed order, this index will get negative value (Fig. 1) Assuming other hypothesis, index will range from 2 to (N / 2) + 1 in proportion to total data mass. On average we will have 2N/5 index. To improve speed we divide the mass of data into fixed sized pieces and implement this in several threads adding the last index to the next index in the next piece according to the value. The last element of a piece is the first element of the next piece.

2	4	6	3	1	9	8	5	7	10	SEQ1
3			-2		-3			2		Index
1	2	3	4	5	6	7	8	9	10	SEQ1
10										Index
10	9	8	7	6	5	4	3	2	1	SEQ1
-10										Index

Figure 1.

3. Worst Case

For any distribution the worst case for implementation of the algorithm to initial 4 elements is described as shown in Table I:

TABLE I.

INITIAL 4 ELEMENTS {A,B,C, D...}				
A < B	B < C	C < D	Index(0)	Index(1)
TRUE	FALSE	TRUE	2	2
TRUE	FALSE	FALSE	2	-2
FALSE	TRUE	FALSE	-2	-2
FALSE	TRUE	TRUE	-2	2

The worst case is the way mergesort begins sort.

4. Average Case

As we see, the worst case splits the distribution into two ordered elements groups. We can then infer that on a random distribution the probability to occur 3 ordered elements is 50%. Therefore a minimal elements group to

define a random distribution would be 2+3 type. This feature is advantageous to mass ordering against Merge Sort algorithm, which in the beginning divide the distribution at each two elements, sort then and merge the resulting groups until the end.

5. Optimal Order

Grouping data according to the smallest groups size optimize non parallel merge routines performance. Nevertheless with multiprocessing the major factor to time optimization is the number of threads on simultaneously work. When a thread begins a large data group process, at the same time many others threads will work on smaller quantities. Mergesort can't use this feature because groups have the same size.

6. Multimerge

The major algorithm routine order two data mass subsets using two threads simultaneously. First thread will merge from beginning until the half of subsets sum, comparing the smallest group elements. Second thread will merge from the end until the other half of the subsets sum, comparing the greatest group elements. (Fig. 2, 3 & 4).

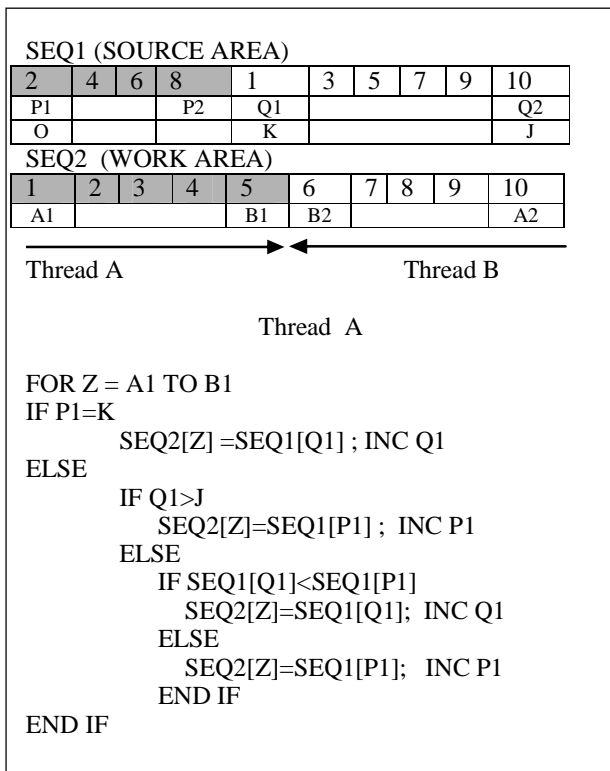


Figure 2 Thread A

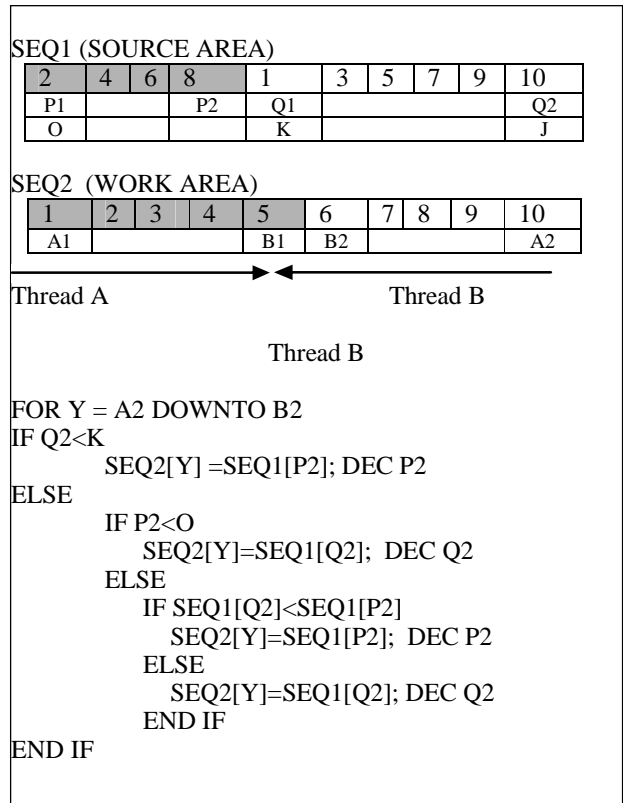


Figure 3 Thread B

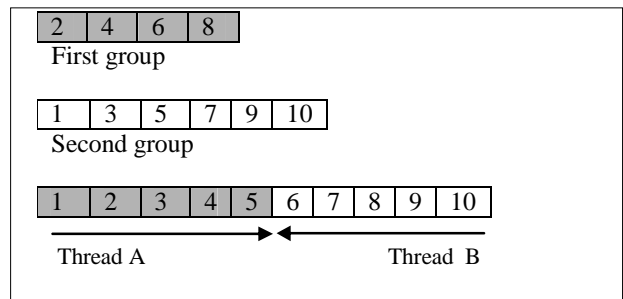
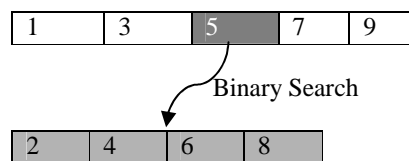


Figure 4

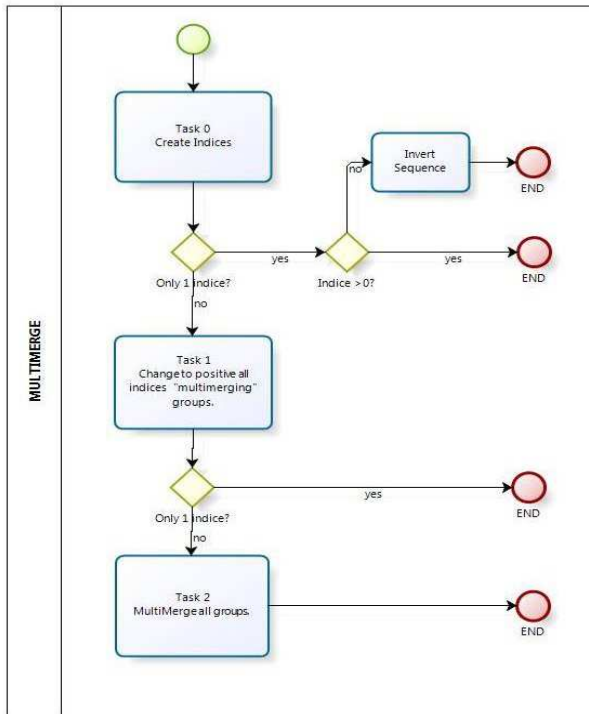
7. Multimerge in Parallel

To increase speed, we need to parallelize the Multimerge. To do this when the sum of the two lists is greater than an arbitrary value, we divide the greater list by two, do a binary search and continue the process until reach that arbitrary value.



8. The Algorithm

The algorithm is composed basically by 3 major Tasks:



The function of Task0 is to identify in the sequence, ordered and reversed order groups, at this time each index is given the status 2. To improve this task the mass of data is divided into groups of $N/m + 1$ elements and parallel threads identify indexes. If they are ordered or reversed order, the next task are not started and the process is redirected to a conclusion.

If the sequence is not ordered Task1 starts. The function of Task1 is to get each group with status 2 and transfer them to the work area, using multimerge and changing the status to 0. Then repeat the procedure taking these groups in the work area and applying multimerge to transfer them to the source area by changing the status to 1. This routine takes the initial mapping to optimize the sorting, because if the sign of the index of adjacent groups are opposing, a position already be set (Table II). Table II

4 ELEMENTS {...,A,B,C, D...}					
$A < B$	$B < C$	$C < D$	Index (6)	Index(7)	Position
TRUE	FALSE	TRUE	2	2	
TRUE	FALSE	FALSE	2	-2	B greatest
FALSE	TRUE	FALSE	-2	-2	
FALSE	TRUE	TRUE	-2	2	B smallest

The Task 1 ends when the last index has its status changed to 1.

If we have more than 1 indice the Task 2 starts and all indices have positive values.

The function of Task 2 is to apply MultiMerge in each group that is in the source area with status equal to 1 and transfer him to the work area by changing the status to 0 and then returning to the source area with status 1.

The algorithm terminates when Task2 transfer data completely ordered to the source area or when there is only one index and its value is equal to N.

To control the flow of threads the algorithm uses a global variable that stores the number of active threads, allowing start of new threads if the maximum limit of threads is not exceeded. The definition of the maximum number of simultaneously threads depending on the characteristics of the hardware.

9. Conclusion

The purpose of this algorithm is:

- identify pre-existing organizations in the mass of data;
- seek the smallest number of iterations for the mass of data;
- distribute the problem in order to fully utilize the processing hardware capabilities;
- consume the smallest possible space allocation, to achieve the solution of the problem in less time.

I hope this text will contribute to the improvement of the processes of sorting and help fellow developers around the world.

10. Acknowledgments

To my wife Michelle for her patience.

11. References

- [1] CORMEN, Thomas H., Introduction to Algorithms, Second Edition, McGraw-Hill Book Company 2001.
- [2] MIT 6.172 Lecture 13 Analysis of multithreaded algorithms http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-172-performance-engineering-of-software-systems-fall-2009/lecture-notes/MIT6_172F09 lec13.pdf
- [3] SZWARCFITER, Jaime Luiz; MARKENZONI, Lilian, (1994) Estrutura de dados e seus algoritmos, Editora LTC 2ª Edição.
- [4] BUCKNALL, Julian; Algoritmos e estruturas de dados com Delphi, Editora Berkeley.

A Study of Memory Access Patterns in Irregular Parallel Codes Using Hardware Counter-Based Tools

Oscar G. Lorenzo¹, Juan A. Lorenzo¹, J.C. Cabaleiro¹, Dora B. Heras¹ Marcos Suárez², and Juan C. Pichel¹

¹Computer Architecture Group, Electronics and Computer Science Dept., Univ. of Santiago de Compostela, 15782 Santiago de Compostela, Spain

²Land Laboratory, Univ. of Santiago de Compostela, 27002 Lugo, Spain

Abstract—This work presents the development of a series of tools to simplify both EARs (Event Address Registers) counters reading and programming in parallel codes. These tools allow EAR counters access in a user friendly workspace. The next tools have been developed: A tool for inserting, in a simple and intuitive manner, the code needed to monitor and program hardware counters in a parallel program. Another tool takes as input the data obtained by the monitored parallel code and shows them in a comprehensive and detailed way. These tools were used to carry out a study of parallel irregular codes and to validate a data reordering technique used to optimize locality of memory accesses in the SpMxV (sparse matrix vector product) problem. Access characterization is one of the main issues dealing with the problem of improving performance of irregular accesses. This is specially true in parallel shared memory platforms.

Keywords: A maximum of 6 keywords

1. Introduction

A tool has been designed for inserting, in a simple and intuitive manner, the code needed to monitor and program hardware counters in a parallel code. Two versions have been implemented. One of them executable from the command line, taking as input a source code parallel program and outputting this same parallel program modified with the monitoring code. The other provides the same functionality using a graphical interface, allowing the user to add the monitoring code gradually.

Another developed tool takes as input the memory access data obtained by the execution of the monitored parallel code and shows them in a graphical and detailed way, while at the same time keeping them clear and user friendly. This tool allows the detail level to be adjusted, so memory accesses can be shown all the way from a global view to detailed, event by event, one.

While there are other similar tools in the market, like *PAPI* [1], *TAU* [2], *Vampir* [3] or *Paraver* [4], our tools are simpler and better oriented to our specific aims.

These tools have been tested during a study of a series of various irregular parallel problems in the *FinisTerae*

supercomputer [5]. In this paper we focus on the use of these tools to validate a data reordering technique [6][7] aimed to optimize the locality of the memory accesses performed by the *SpMxV* code. Although this work has been carried out in the *FinisTerae*, these tools can be generally applied to any other hardware counter enabled *Itanium2* [8] processor based environment, both monoprocesor and multiprocessor.

The *Itanium2* [8] processor family implements a special kind of hardware counters called EARs. These counters offer information about events related to memory accesses at virtual addressing level. This information includes, among others, TLB misses and cache misses at different levels, with their access latency, for each virtual memory address. The cache hierarchy of this processor is shown in Figure 1. This information might be of great value to the programmer, but its access may be complex and tedious. In this work we present different tools to simplify the use of these hardware counters.

EARs can be configured to capture only those events in which we might be interested. For cache miss events both the read resolution minimum latency, with values from 4 to 4096 cycles, and the cache type, either data or instruction, can be considered. Here we focus in data cache misses, usually with low latencies, as to capture the greatest number of events possible.

To carry out the PMU programming as well as to obtain of execution data both the *libpfm2* [9] library and the *perfmon2* [9] communication interface have been used.

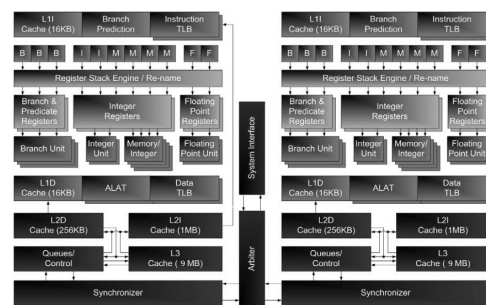


Fig. 1: The Itanium2 Montvale processor.

2. The code insertion tool

This tool aims to solve the problem of adding the necessary code to work with EAR counters in a parallel program in the most automatic way. The user must be allowed to name the event to capture or the point in the code from where to start monitoring, among other considerations, but this tool reduces greatly the amount of work needed to do this task. For each miss event relevant information is captured: the virtual memory address that was accessed, the accessing instruction memory address, the core where it took place and an additional item, the operation's latency in cache miss events.

This tool adds code automatically in three points in the parallel code (see Figure 2):

- **Previous Code:** This code must come before both counter programming and reading. It is made of library inclusion directives, constant and variable declaration, and the global procedures to be used later (mainly those regarding EARs reading).
- **Begin Code:** This code comes exactly before monitoring begins, and must be inside the same parallel region as the code to be monitored. It is made of both *libpfm* initialization and PMU counters programming. It ends issuing the order to start monitoring.
- **End Code:** This code takes care of ending monitoring, so it must be added just after the section to be measured. It processes any information remaining in the sampling buffer.

```

PARALLEL PROGRAM TO MEASURE
#include <omp.h>
#include <stdio.h>

PREVIOUS CODE

main(){
    double A[1000], B[1000];
    int i;

    #pragma omp parallel
    {
        BEGIN CODE

        #pragma omp for
        for(i=0; i<1000; i++){
            A[i] = B[i]*15;
        }

        END CODE

    } //end parallel

    ...

} //end program
    
```

Fig. 2: Added code.

3. The visualization tool

The visualization tool main functionality is to classify captured events in categories according to their memory

address. It then shows them jointly in a histogram delimited by the initial and final addresses of the studied virtual memory range as seen in Figure 3. This way it allows the user to get a general view of memory access patterns in her parallel program. This histogram can be modified by the user, for example, by filtering events by their instruction or core, or by changing the number and size of the categories, as to raise or lower the detail level (see Figure 4).

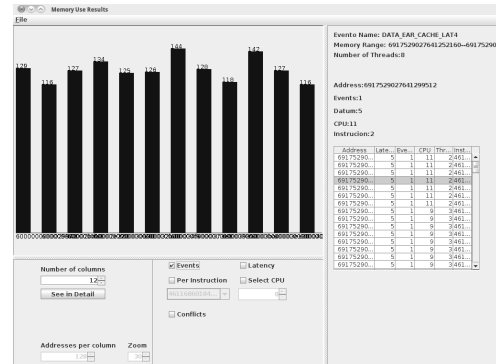


Fig. 3: Visualization tool main histogram.

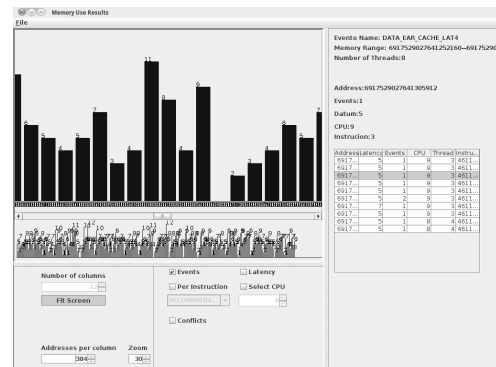


Fig. 4: Visualization tool detailed histogram.

4. A case study: locality optimization technique

A locality optimization technique [6][7] was used to test the above tools. It consists of reorganizing the data guided by a locality model instead of restructuring the code or changing the sparse matrix storage format. Unlike other existent locality models that predict in a precise way the data movement among the levels of the memory hierarchy, our model is able to characterize, sacrificing accuracy, the trend of this movement in general terms. In particular, locality is evaluated using a distance function that depends on the number of entry matches. Considering accesses to the sparse matrix by rows, the number of entry matches between any pair of rows is defined as the number of non-zero elements in

the same column of both rows. The model provides a value, noted as total distance, that is inversely proportional to the locality of the accesses performed by the sparse matrix.

The goal of the data reordering technique is to increase the grouping of non-zero elements in the sparse matrix pattern that characterizes the irregular accesses and, as a consequence, improve the locality in the execution of the *SpMxV* code. The reordering technique permutes windows of locality instead of individual rows or columns of the matrix. A window of locality is a set of w consecutive rows (or columns) of the matrix between which there is a high probability of data reuse when executing the sparse matrix code.

The method searches for a permutation of windows of locality that minimizes the total distance of the matrix. In this way, the problem of locality improvement is formulated as a classic NP-complete optimization problem, and we solve it as a graph problem using its analogy to the traveling salesman problem. The appropriate order of rows and columns of the matrix is given as a permutation vector.

In order to select the window size (w) [7], two types of windows of locality are considered: fixed and variable. In the variable-size case, the criterion to decide if two consecutive rows/columns are included within the same window depends on the locality estimation made by the model. In particular, the distance between these rows/columns must be lower than a established threshold. In our studies we conclude that windows of $w = 1$ and $w = \text{variable}$ are the best choices in terms of performance.

4.1 Experimental results

The data reordering technique has been applied to the matrices shown in Figure 5 using both window sizes. These reordered matrices should present a higher data locality for each thread individually, which would mean that fewer memory addresses should be shared between them.

In the *Itanium2* architecture, floating point operands bypass the level 1 data cache, so reading them always causes a cache miss [8]. Since the matrix data are floating-point values the EAR counters can be used to sample among all memory accesses. Using the tools previously commented data regarding the *SpMxV* memory accesses have been easily collected at a high sample rate.

In Figure 6 the number of total read conflicts caused by different threads accessing the same memory address are shown. These are normalized for each pair matrix-number of threads to the results regarding the unordered matrix. It shows how both window sizes ($w = 1$ and $w = \text{variable}$) are effective for most matrices, and more effective as the number of threads grows. Only for matrices *bcstkt29* and *memplus* results do not show an improvement with low number of threads. This is because the pattern of both matrices already presents high locality, and as such it cannot be improved at all by the reordering technique.

5. Conclusions

The tools here developed greatly simplify obtaining and studying data from the EAR counters present in the *Itanium2* processors. With these, data memory accesses can be traced during a program's execution. We study shared memory programs using *OpenMP*. These tools call for some familiarity using PMU hardware counters, but they nonetheless simplify adding monitoring code. Since this code is user editable they can be adapted for use in any number of environments, architectures or codes. The data visualization tool allows for an easy statistical study of the captured events, by offering the most important functionalities related to working with counter data.

It has been shown that the information obtained is useful to model the execution of a parallel program, by studying its memory access patterns. Using this information it is easy to see, for example, which threads have a greater workload, relative to their input data, since it determines the number of memory accesses and their locality.

Using these tools a reordering technique has been validated to show its benefits. In fact, results show locality is greatly improved by this reordering techniques.

Acknowledgment

This work has been partially supported by the Ministry of Education and Science of Spain, FEDER funds under contract TIN 2010-17541 and by the Xunta de Galicia (Spain) under contract 2010/28 and project 09TIC002CT. The authors also wish to thank the supercomputer facilities provided by CESGA.

References

- [1] <http://icl.cs.utk.edu/papi/>, Performance Application Programming Interface (PAPI).
- [2] S. S. Shende and A. D. Malony, "The Tau parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, Summer 2006. [Online]. Available: <http://hpc.sagepub.com/content/20/2/287.abstract>
- [3] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of mpi resources," *Supercomputer*, vol. 12, pp. 69–80, 1996.
- [4] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, "Dip: A parallel program development environment," in *Euro-Par'96 Parallel Processing*, ser. Lecture Notes in Computer Science, L. Boug  f, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds. Springer Berlin / Heidelberg, 1996, vol. 1124, pp. 665–674, 10.1007/BFb0024763. [Online]. Available: <http://dx.doi.org/10.1007/BFb0024763>
- [5] http://www.cesga.es/Galicia_Supercomputing_Center.
- [6] J. C. Pichel, D. E. Singh, and J. Carretero, "Reordering algorithms for increasing locality on multicore processors," in *Proc. of the IEEE Int. Conf. on High Performance Computing and Communications*, 2008, pp. 123–130.
- [7] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera, "Increasing data reuse of sparse algebra codes on simultaneous multithreading architectures," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 15, pp. 1838–1856, 2009.
- [8] *Dual-Core Update to the Intel Itanium 2 Processor Reference Manual*, download.intel.com/design/Itanium2/manuals/30806501.pdf.
- [9] *Perfmon2 monitoring interface and Pfmmon monitoring tool*, <http://perfmon2.sourceforge.net>.

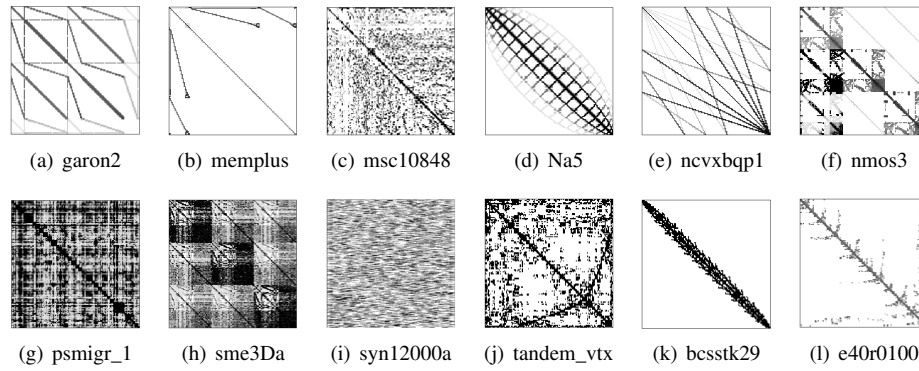
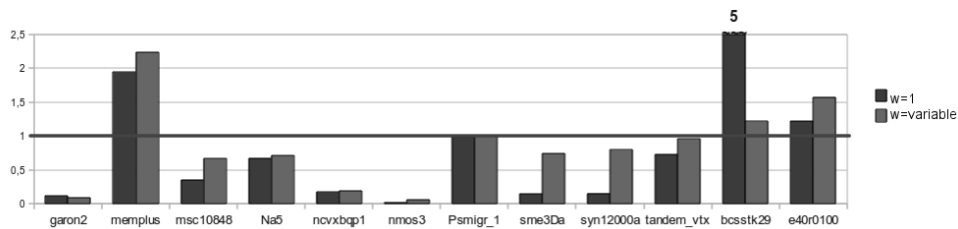
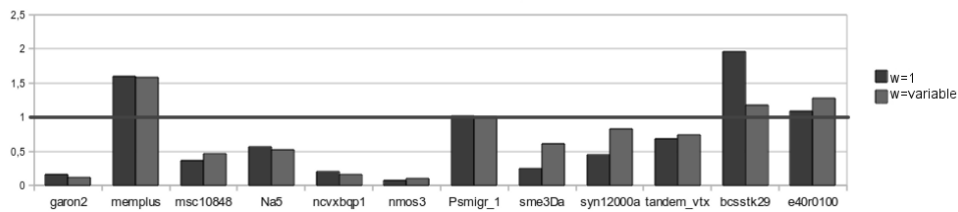


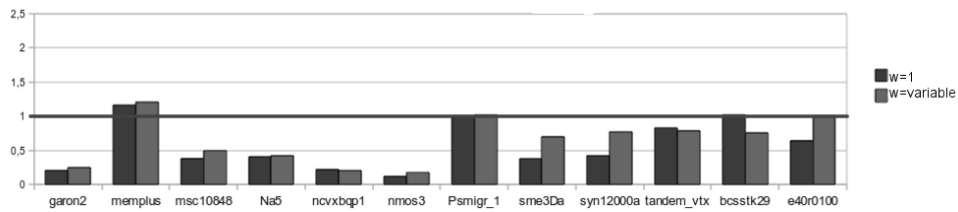
Fig. 5: Matrix patterns before reordering.



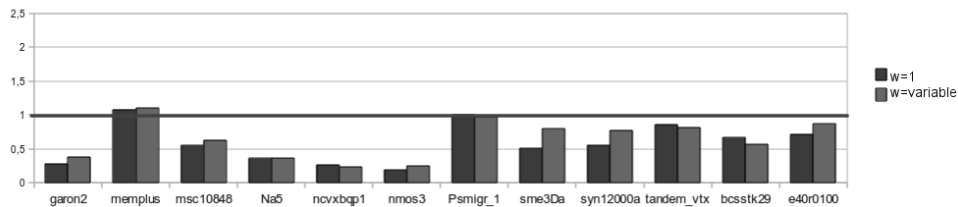
(a) Total Conflicts Fraction - 2 threads



(b) Total Conflicts Fraction - 4 threads



(c) Total Conflicts Fraction - 8 threads



(d) Total Conflicts Fraction - 16 threads

Fig. 6: Conflict reduction after applying both reordering techniques, normalized to the unordered matrix. These results have been obtained using the hardware counter based tools.

VLSI Parallel Sorter Architecture for Streaming Data

Dongjae Song, Kyoung Kun Lee, Soongyu Kwon, and Jong Tae Kim

School of Information and Communication Engineering,
Sungkyunkwan University,
Suwon, Gyeonggi-do, South Korea

Abstract - Data sorting is used indispensably in computer system and affects performance of entire processing. Recently, its importance is increased in many areas from database server system to embedded computer system. Data sorting algorithms and implementation methods have been conducted in many ways that reduces complexity for operation purposes, along with hardware/software structures. In this paper, we propose new sorting architecture that uses 8-way merge sort algorithm to sort streaming data from database. By using pipelined merge sort algorithm, we can organize parallel sorter architecture. Through functional simulation, we analyze relation between amount of data and required clock cycle to sort the data. Also, we can calculate size of memory to implement sorter architecture.

Keywords: VLSI architecture, hardware sorter, parallel sorter, merge sort

1 Introduction

In most computer system and database system, data sorting is basic and indispensably used operation. Therefore, data sorting takes important part in capability of entire system. Its importance is getting bigger from database server to embedded computer system [4]. Because of its importance, many experiments have been conducted across the board on system before to suggest algorithms that reduces complexity and sorting time for operation purposes, along with hardware/software structures [1][3][5]. Through them, many parts are improved and many algorithms were developed. However, data sorting is still one of the operations that take most time. That is why it is important to see how sorting system, which accompanies sorting application, is constructed. Data sorting algorithms and implementation methods have been conducted in many ways. For example, at algorithmic aspect, quick sort, heap sort or merge sort algorithm can be used. And sorter can consists of hardware only or hardware/software hybrid method [6]. These implementation of the sorter has advantages and disadvantages of each are quite different.

Our objective is to reduce unnecessary time in sorting streaming data that comes through input and develop VLSI architecture of hardware sorter that outputs sorted data. Using

8-way pipelined merge sort algorithm, we design hardware sort unit model using VHDL. Entire sorting system is developed by connecting these sort units. Then, simulate system for verification and performance analysis. Through functional simulations, we derive a simple formula for relationship between amount of data and required clock cycle to sort data.

This paper is comprised as following. Section 2 describes many hardware sorter architecture and pipeline merge sort algorithm. Section 3 describes our approaches to materializing VLSI architecture of hardware sorter. Section 4 shows result of experiment. Section 5 determines conclusion that arises from result of experiment.

2 Related Work

2.1 Pipeline Merge Sort Algorithm

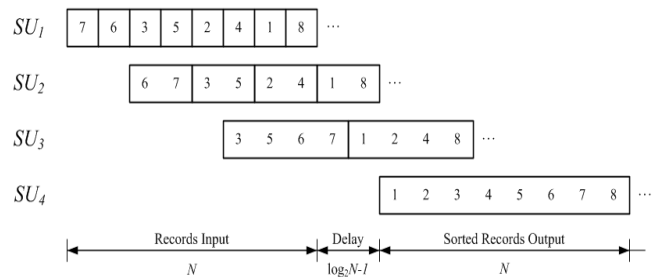


Figure 1 Pipeline merge sort algorithm

Merge sort Algorithm performs multiple levels of merge and compare operation when input is received. Essentially $\log_2 n$ steps are done and applied to the data in 2-way merge sort. This process is done in typical time so each merge step takes $O(n)$ time. Therefore, total operational time costs get $O(n \log n)$ complexity. In case of 2-way merge sort, the first step is to merge 1 record. Partially sorted records are called strings. The second step merges strings with size of two. m -th step merges string size of 2^{m-1} . Looking at example shown in Figure 1, SU_1 (Sort Unit) takes in a series of records and outputs one arranged string from two records. 7 is greater than 6 so 6 comes first and 7 is put at next clock cycle. During the process, next records 3 and 5 are computed. SU_2 , which

already has input values of string (6, 7), compares first record of next string 3 with 6 and since 3 is smaller than 6, 3 is output first. Next calculation compares smallest values from remaining records of strings. Through operation, if N records are input, it takes $2N + \log_2 N - 1$ time to finish sorted output. In K -way merge sort, through SU_1 , each record is inputted, K records are sequentially ordered to one output string. K strings (K^2 Records), which is the output of SU_1 , becomes input of SU_2 . To express it in general terms, if one input of SU_m is a string which consists of K^{m-1} records, output becomes a string which has K^m records.

2.2 Hardware Sorter Architecture

Table 1 Comparison of different sorter architectures

Architecture	Throughput	Input constraint
Parallel	2^l	2^l
Semi-Parallel	2^{l-1}	2^l
Iterative	$< 2^l \times M / ((l^2 - l + 4) \times 2^{l-2} - 1)$	$2M$
Pipeline	1	1

Hardware sorter is differentiated in 4 ways depending on how architecture is formed (parallel, semi-parallel, iterative, pipeline). Table 1 shows comparison of throughput and input constraint of different types of sorter. Looking at the throughput, full-parallel show fastest throughput but it takes that much area and gates [7]. M is a degree of parallelism so it lowers hardware efficiency when taking in large values. Therefore, the throughput listed in Table 1 for serial sorters only represents an upper-bound.

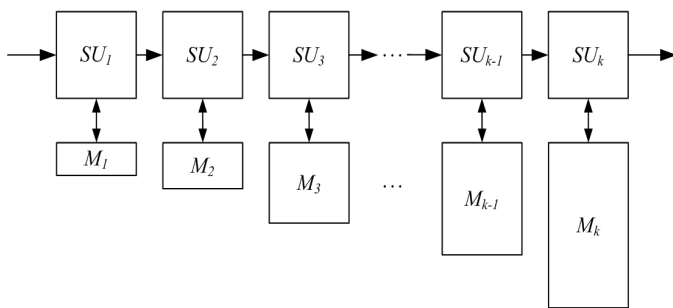


Figure 2 Hardware sorter using external local memories

Hardware sorters in [5] and [6] use local memory which lies outside of needed memory as shown in figure 2. This method uses interface block to control memory. Since this method put memory outside and queue inside of sort unit, complexity of sort unit is reduced. However, additional design is needed for memory interface, which stores data to memory and reuses data.

3 Approach

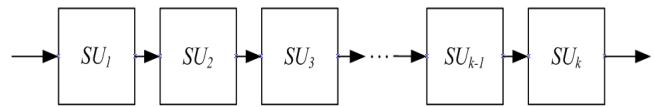


Figure 3 Configuration of suggested hardware sorter

Suggested architecture is materialized by reducing time delay that is addition to time required for merge sort algorithm. Figure 3 is a block diagram that shows structure of hardware sorter. Hardware sorter uses these sort units that are connected in a straight line. In 8-way merge sort, to order 8^k records, it needs k sort units. One sort unit, shown in Figure 4, is comprised of input block, which takes in streaming data from database, disk, or output from previous step, and sort processor block, which compares input data and outputs finished data. Data width is 100-byte (10-byte key) which can send one record in one clock cycle. In our approach, we can extend data sorting capability by connecting sort unit in a straight. Using this method, once clock frequency is determined, we can calculate capability and elapsed time of sorter.

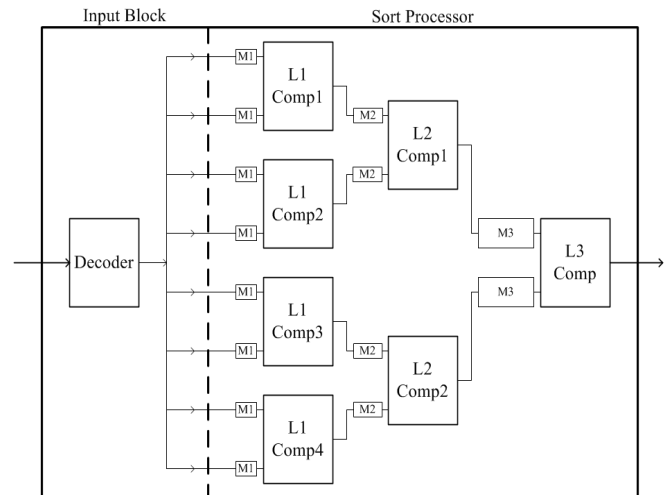


Figure 4 Sort unit block diagram

Our sort processor puts memory inside and seven comparators are comprised in tree structure as shown in Figure 4. Eight inputs of the sort processor are taken in partially ordered strings from input block. Records in input string are stored sequentially in memory. Operation is started when first record from second memory comes in and output comes out after comparison. Input block takes in data of disk or values from previous sort unit to $M1$ of sort processor. At this time, according to SU_k , data which is sent $M1$ is controlled. For example, in SU_1 , one record is sent to each $M1$ of sort processor. In case of SU_2 , eight records are sent. This

indicates that memory of SU_{k+1} needs K times the capacity of memory of SU_k . To sort one million records, we use seven sort units through SU_1 to SU_7 . To take care of 2^{20} data, 8-way sort is used between SU_1 to SU_6 . 4-way sort is used for step SU_7 .

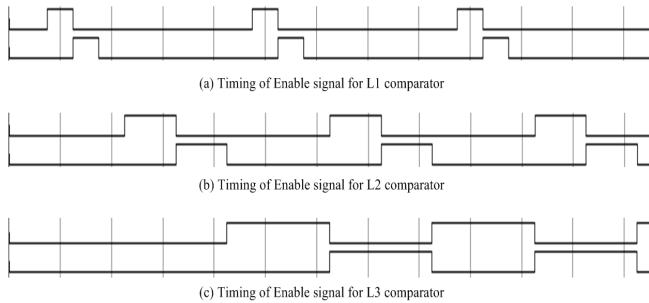
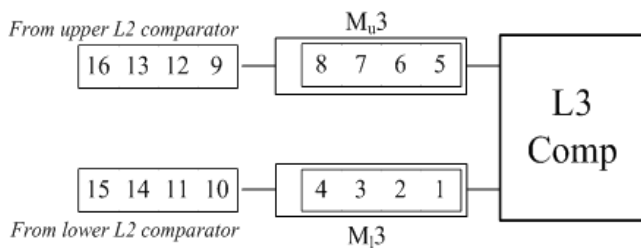
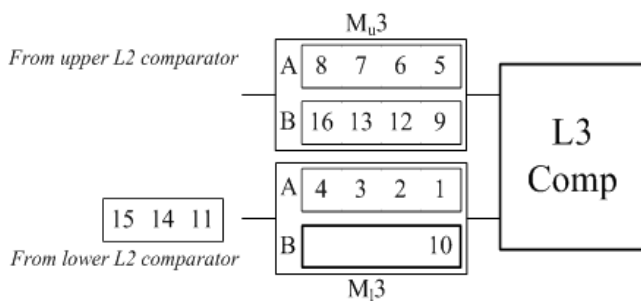


Figure 5 Timing of each comparator enable signal

Each comparator operates while some of input strings are stored. Therefore, comparator needs memory equal to string size. It also needs operational time to finish comparing input string. Figure 5 shows write enable (WE) signal when string input in comparator. Figure 5(a) and (b) are WE signals of $L1$ and $L2$ comparators. It has no problems since there is plenty of time in clock cycle between first WE signal and next one. Figure 5(c) shows WE signal of $L3$ comparator that comes in consecutively. In this case, using memory of $L1$ and $L2$, two strings can be overwritten.



(a) Data overwriting occurs at L3 comparator



(b) Avoid data overwriting using additional memory

Figure 6 Worst case scenario at L3 comparator

Figure 6(a) shows data overwriting case. If all input records of M_l3 ($M3$ lower) is smaller than all of M_u3 ($M3$ upper), next input sorted strings of M_u3 from upper $L2$

comparator will overwrite current M_u3 records. To avoid this malfunctioning operation, memory size of $L3$ comparator is twice as big as size of input string.

4 Simulation Results

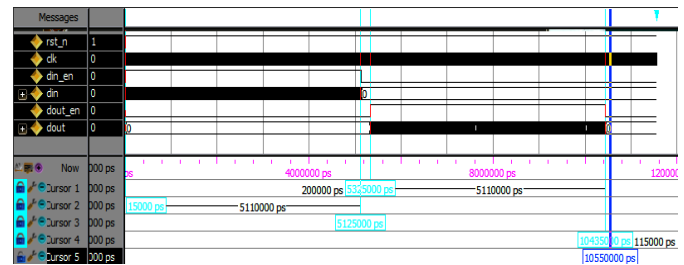


Figure 7 Hardware sorter functional simulation

Using approach of previous section, RTL-code of hardware sorter is developed with VHDL. Figure 7 is result of functional simulation using hardware sorter that was designed with approach of previous section. Sorter that was simulated is designed through SU_1 to SU_3 . 512 streaming data was arranged through input. Input was put in 512 clock cycles. After 20 clock cycle wait, 512 clock cycles were needed for output, which confirms pipeline structure with throughput of 1. Simulation clock cycle was 10ns. 512 records were input. After 20 clock cycles, we can see 512 sorted output strings. With design of hardware sorter, one clock cycle delay happens each time it goes through input block and each step of comparator within sort processor. Therefore, when N number of data is input as records, delay that happen when it processed to i level comparator of k -th sort unit is detailed below.

$$\begin{aligned}
 \text{Total Delay Time (clock cycles)} &= \text{input records} + \text{delay} + \text{output records} \\
 &= N + \lceil \log_2 N + 4(k-1) + i \rceil + N
 \end{aligned}$$

With this relationship, Table 2 is arrangement of sort unit with its input data, delay time, and total delay time. In this table, capability means total records can be sorted by connecting SU_1 to SU_k in a straight line. Delay time means clock cycle difference between end of input records and start of output records. Similar context with delay time, total delay is time takes start of input records to end of output records.

It can be figured out that as input data gets larger, ratio of delay time, excluding time for input and output, gets smaller. Ideally, with 100MHz clock input, data line width being 100-byte, and 100-byte is being input for each clock cycle, a million data can be sorted in 0.02 seconds.

Table 2 Capabilities and delays for each sort unit

Sort Unit	Capability (records)	Delay (cycles)	Total Delay (cycles)
SU_1	8	6	22
$SU_1 \sim SU_2$	64	13	141
$SU_1 \sim SU_3$	512	20	1,044
$SU_1 \sim SU_4$	4,096	27	8,219
$SU_1 \sim SU_5$	32,768	34	65,570
$SU_1 \sim SU_6$	262,144	41	524,329
$SU_1 \sim SU_7$	1,048,576	46	2,097,198

Table 3 Memory size of each sorting unit

(unit : 100-byte)

Sort Unit Memory	SU_1	SU_2	SU_3	SU_4
M1	1	8	64	512
M2	2	16	128	1024
M3	8	64	512	4096
Sort Unit Memory	SU_5	SU_6	SU_7	
M1	4096	32768	262144	
M2	8192	65536	524288	
M3	32768	262144	-	

Table 3 shows memory size of each sort unit for sorting one million(2^{20}) records. Width of each record is 100-byte include 10-byte key. To sort one million records, 8-way merge sort is used between SU_1 to SU_6 and 4-way merge sort is used for SU_7 . Total memory required is about 628.6MB to sort one million records.

5 Conclusion

Sorting is important and necessary part which can affect system performance in the computer system. Its importance is getting larger from database server to embedded computing system. This paper used 8-way merge sort to design architecture of hardware sorter in order to sort streaming data in parallel way. Merge sort is naturally pipelined algorithm. We designed sort processor using merge sort algorithm. Entire system consists of sort units which connected in straight line. This straight line architecture allows each sort unit operate in parallel way. Also, this architecture is scalable for amount of data. Depending size of data to be sorted, we know how to change structure of sorter. Also we can calculate the number of sort units and amount of memory needed. Through simulation, we checked out delay time of suggested hardware sorter is different depending on size of data. We confirmed that delay time is reduced in terms of ratio as more data were sorted, which took advantage of positive trait of pipeline structure.

6 References

- [1] A.A. Colavita, et al, "SORTCHIP : A VLSI Implementation of a Hardware Algorithm for Continuous Data Sorting," *IEEE Journal of Solid-State Circuits*, Vol. 38, pp. 1076-1079, No.6, Jun. 2003.
- [2] Anon , et al, "A Measure of Transaction Processing Power," *Datamation*, 31(7):112-118, 1985.
- [3] M. Bednara, et al, "Tradeoff Analysis and Architecture Design of a Hybrid Hardware/Software Sorter," *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, Jul. 2000.
- [4] R. Marcelino, et al, "A Comparison of Three Representative Hardware Sorting Units," *IECON '09. 35th Annual Conference of IEEE*, Nov. 2009.
- [5] S. Azuma, et al, "DIAPRISM Hardware Sorter," Sort Benchmark, 2000.
http://sortbenchmark.org/Y2000_Datamation_DiaprismSorter.pdf
- [6] S. Fushimi, et al. "GREO : A Commercial Database Processor Based on A Pipelined Hardware Sorter," *ACM SIGMOD '93*, vol.22, No.2, pp.449-452, 1993.
- [7] Yun-Nan Chang, "Digit-Serial Pipeline Sorter Architecture," Springer, *Journal of Signal Processing Systems*, Vol 61, Issues 2, Nov. 2010.

