THESIS

EFFICIENT RUNTIME AND ALGORITHMS FOR PARALLEL PROGRAMMING

UNDER GRID ENVIRONMENT

THEEWARA VORAKOSIT

GRADUATE SCHOOL, KASETSART UNIVERSITY

2006

THESIS APPROVAL

GRADUATE SCHOOL, KASETSART UNIVERSITY

Doctor of Engineering (Computer Engineering)

DEGREE

Computer Engineering | Computer Engineering

FIELD | **DEPARTMENT**

TITLE: Efficient Runtime and Algorithms for Parallel Programming under Grid Environment

NAME: Mr. Theewara Vorakosit

THIS THESIS HAS BEEN ACCEPTED BY

..... **THESIS ADVISOR**

(Assistant Professor Putchong Uthayapos, Ph. D.)

..... **COMMITTEE MEMBER**

(Assistant Professor Arnon Rungsawang, Ph. D.)

..... **COMMITTEE MEMBER**

(Assistant Professor Kemathat Vibhatavanij, Ph.D.)

..... **DEPARTMENT HEAD**

(Assistant Professor Kemathat Vibhatavanij, Ph.D.)

APPROVED BY THE GRADUATE SCHOOL ON .....

..... **DEAN**

(Associate Professor Vinai Artkongharn, M.A.)

THESIS

EFFICIENT RUNTIME AND ALGORITHMS FOR PARALLEL PROGRAMMING
UNDER GRID ENVIRONMENT

THEEWARA VORAKOSIT

A Thesis Submitted in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Engineering (Computer Engineering)
Graduate School, Kasetsart University
2006

ISBN 974-16-2764-5

Theewara Vorakosit 2006: Efficient Runtime and Algorithms for Parallel Programming under Grid Environment. Doctor of Engineering (Computer Engineering), Major Field: Computer Engineering, Department of Computer Engineering. Thesis Advisor: Assistant Professor Putchong Uthayopas, Ph.D. 102 pages.
ISBN 974-16-2764-5

This dissertation addresses many challenges encountered in the building an efficient MPI run-time library on a Grid system. The first problem is how to enable the MPI run time library to fully utilize all Grid nodes under an environment where majority of the nodes are hidden behind the gateway nodes. The second problem is how to efficiently multicast MPI messages when the topology of the network is dynamic and highly heterogeneous.

For the first problem, this dissertation proposes a new way of modeling a Grid system as a set of virtual clusters. Using this model, a mathematical analysis has been provided to prove that an efficient and valid grid wide routing algorithm exists. In addition to addressing the routing problem, this dissertation also shows how to formulate Grid multicasting problem into a combinatorial optimization problem. Then two new multicast algorithms called GADT (Genetics Algorithm based Dynamic Tree) and LPBF (Longest Parallel Branch First) are proposed as the solutions. The first algorithm, GADT, uses evolutionary computing approach to generate a near optimal multicast schedule. But the algorithm requires a moderate run time which make it more suitable as a tool to find the maximum performance attainable than being used in an MPI runtime library. In contrast, LPBF is designed for a fast and efficient algorithm to be implemented in an MPI runtime library. LPBF employs an intelligent message scheduling technique to exploit an inherent communication and computation overlapping. Thus, the total transmission time of a multicast operation is reduced substantially.

The proposed algorithms have been tested using the simulation and the implementation on a grid system. An experimental Grid-enable MPI library named MPITH has been developed for that purpose. A campus grid in Kasetsart University, Thailand, is used along with the international Grid operated by PRAGMA project. It has been found that the proposed multicast algorithms perform efficiently in both simulation and on a real Grid test-bed system. Moreover, the proposed routing algorithm allows MPI applications utilize all the nodes that belong to a test Grid system with a very low performance penalty. In summary, the contribution of this dissertation is to propose the model and algorithms that can improve the performance of MPI run time library on the Grid systems. This will enable a large class of MPI applications to be ported and run on a large scale grid system efficiently.

_____ _____ __ / __ / __

Student's signature Thesis Advisor's signature

ACKNOWLEDGEMENTS

I would sincerely like to acknowledge the efforts of many people who contributed to the research and this dissertation.

First, I would like to thank to my parents, who take care of everything for me. I would like to dedicate this dissertation for them.

I would like to gratefully thank my dissertation advisor, Assistant Professor Dr. Putchong Uthayopas, who gives me valuable ideas and knowledge since I was an undergraduate student.

I would like to thank Assistant Professor Dr. Arnon Rungsawang, Assistant Professor Dr. Kemathat Vibhatavanij and Dr. James Edward Brucker for their time and effort. In addition, I would like to extend my gratitude towards many lecturers in the Department of Computer Engineering, Faculty of Engineering, Kasetsart University, who contributeed directly or indirectly to this research.

I would like to thank my friends, Krit Srisarn, Sugree Phatanapherom, Thisana Thitisakdiskul, Porjai Wongprawmas, Thara Angskun, Somsak Sriprayoonsakul, and Kasom Koht-arsa for many suggestions. For my colleagues in Scalable Cluster Environment (SCE) project, thank you for your good collaboration and perfect teamwork. I would like to thank my friends from High Performance Computing and Networking Center for the stimulating discussions and help. I would like to thank to Ms. Pakjira Jaiyim for her assistance in many ways.

Finally, I would like to thanks to KURDI grant SRU who supported this research. The test bed used in this research is a part of PRAGMA Grid facility.

Theewara Vorakosit
October 2006

TABLE OF CONTENTS

Page

TABLE OF CONTENTS i
LIST OF TABLES ii
LIST OF FIGURES iii
INTRODUCTION 1
- Problems 1
- Objective 3
- Research Scope 3
- Background Theory 3

LITERATURE REVIEW 15
- Wide Area Collective Communication Algorithms 15
- MPI Implementations 17

MATERIALS AND METHODS 20
- Materials 20
- Methods 21
- A Virtual Cluster Model 22
- Proposed Routing Discovery Algorithm 28
- Multicast Operations Model 31
- Genetic Algorithms-based Dynamic Tree Algorithm 33
- Longest Parallel Branch First Algorithm 37
- Tree Simulator 40
- MPITH Design and Implementation 42
- Development Experience 51

RESULTS AND DISCUSSION 53
- Multicast Algorithms Performance 53
- TreeSim Performances 62
- MPITH Performances 66

CONCLUSION 80
LITERATURE CITED 81
APPENDIX MPITH Reference Manual 86
- MPI Namespace Reference 86
- MPITH Namespace Reference 90
- MPI::Comm Class Reference 91
- MPI::Datatype Class Reference 95
- MPI::Group Class Reference 96
- MPITH::Host Class Reference 97
- MPI::Intracomm Class Reference 98
- MPI::Op Class Reference 100
- MPITH::Process Class Reference 100
- MPI::Status Class Reference 101

CURRICULUM VITAE 102

LIST OF TABLES

Table		Page
1	Features comparison between MPI implementations	19
2	An example of a routing table of three virtual clusters under the proposed virtual cluster model	30
3	An example of a process mapping	30
4	An example of a node mapping	30
5	A multicast order	32
6	The number of node for two clusters	54
7	PGA pack parameters of test environment	55
8	Total transmission time of eight nodes, a symmetric bandwidth	55
9	Total transmission time of eight nodes, an asymmetric bandwidth	55
10	Normalized total transmission time of eight nodes, symmetric bandwidth	57
11	Normalized total transmission time of eight nodes, asymmetric bandwidth	57
12	A test bed configuration	66
13	Test applications	74
14	Speedup of heat transfer application	75
15	Normalized runtime of heat transfer application	75

LIST OF FIGURES

Figure		Page
1	A cluster architecture	9
2	The core elements of the OGSA	11
3	A 3-dimensional cube for binary DNA string of length 3	12
4	Magi cluster	20
5	An example of a Grid environment	23
6	A Grid graph derived from Figure 5	23
7	A bandwidth matrix for Figure 6	23
8	Connectivity graph of Figure 5	24
9	A DCG derived from Figure 8	25
10	Virtual clusters in Gd	25
11	Vc[1] in Gd	26
12	Intermediate step of VC merging	26
13	Final VC merging	27
14	A virtual cluster graph of Figure 10	28
15	An example of routing scenario	31
16	A multicast topology	32
17	An example of multicast schedule	33
18	A GADT DNA encoding from Figure 17	33
19	A child table generated from Figure 18	34
20	An example of invalid DNA	34
21	A multicast schedule generated from Figure 20	34
22	A GADT algorithm	37
23	A multicast tree obtained from the first two steps of LPBF algorithm	38
24	Branch times	38
25	An LPBF schedule	39
26	A TreeSim architecture	40
27	An evaluator’s algorithm	41
28	The MPITH architecture	42
29	A process startup sequence	46
30	An MPITH message format	47
31	The incoming buffer	47
32	The outgoing buffer	48
33	The I/O module architecture	49
34	Total transmission time of eight nodes, symmetric bandwidth	56
35	Total transmission time of eight nodes, asymmetric bandwidth	56
36	Normalized total transmission time of eight nodes, symmetric bandwidth	58
37	Normalized total transmission time of eight nodes, asymmetric bandwidth	58
38	Total transmission time of eight clusters, symmetric bandwidth	59
39	Total transmission time of eight clusters, asymmetric bandwidth	60
40	Total transmission time of 128 nodes, symmetric bandwidth	61

LIST OF FIGURE (CONT'D)

Figure		Page
41	Total transmission time of 128 nodes, asymmetric bandwidth	61
42	Simulation time for eight nodes, symmetric bandwidth	62
43	Simulation time for eight nodes, asymmetric bandwidth	63
44	Simulation time for eight clusters, symmetric bandwidth	64
45	Simulation time for eight nodes, asymmetric bandwidth	64
46	Simulation time for 128 nodes, symmetric bandwidth	65
47	Simulation time for 128 nodes, asymmetric bandwidth	66
48	The test bed topology	67
49	A performance of MPI_Send/MPI_Recv on Magi cluster	67
50	A comparison between single and double hops MPI_Send/MPI_Recv	68
51	A performance of MPI_Bcast in a cluster environment	69
52	A performance of MPI_Bcast in a Grid environment	70
53	A performance of MPI_Reduce	71
54	A performance of MPI_Scatter	72
55	A performance of MPI_Gather	73
56	Runtime of heat flow application	74
57	Runtime of Gaussian elimination	76
58	Speed up of Gaussian elimination	76
59	Normalized runtime of Gaussian elimination	77
60	Runtime of VaR application	78
61	Speed up of VaR application	78
62	Normalized runtimes of VaR application	79

EFFICIENT RUNTIME AND ALGORITHMS FOR PARALLEL PROGRAMMING UNDER GRID ENVIRONMENT

INTRODUCTION

Problems

A Grid system (Foster and Kesselman, 2003) provides an infrastructure for sharing computational resources across multiple organizations. A wide variety of high performance applications benefit from a Grid system by running many processes cooperatively on the Grid. Many programming models (Lee and Talia, 2003) have been developed to harvest the enormous computing power of a Grid system such as Grid RPC, task parallelization, and message passing. The message passing model provides a uniform programming paradigm that scales from a single shared memory parallel computer up to a large, distributed Grid environment. The Message Passing Interface (MPI) (Snir *et al.*, 1998; Gropp *et al.*, 1998) provides a standard API and set of services for this. MPI was developed for parallel computers and clusters; extensions to a distributed Grid environment are an area of on-going research. The master-slave and parameter sweep application are examples of MPI application that can utilize resources in Grid system because these types of applications have a moderate communication volume.

The Grid environment raises several issues that complicate message passing and impairs application performance (compared with a cluster or parallel computer), such as high latency, lower bandwidth, security and authentication, problems of collective communications, and routing issues. Among of these problems, this dissertation proposes solutions to two challenging problems found in a Grid system: closed cluster environment, and efficient MPI collective operation.

The first problem, a closed cluster environment, occurs because a Grid system contains many *closed clusters* of computing nodes that are not directly reachable by other nodes. These clusters typically use private IP addresses with a single Network Address Translation (NAT) gateway providing internetwork connectivity. If MPI implementations require that all participating nodes must be reachable by each other, hence, only gateway nodes of closed clusters can be used in most Grid MPI applications. This substantially reduces the amount of computing power available to Grid users. This dissertation proposes a systematic model of logical interconnection between nodes called a *virtual cluster (VC)* (Vorakosit and Uthayopas, 2005a). In this VC model, logical connections between nodes are modeled as a graph named *directed communication graph (DCG)*. Nodes in a Grid are grouped together to a *virtual cluster*. Routes are created from a virtual cluster graph and are used to create routes between MPI processes at runtime.

In addition to the closed clusters problem, MPI collective communications also greatly influence application performance on a Grid system. The bandwidth of network links among clusters varies significantly, as may the speed of intra-cluster

communications between nodes to a few Megabits per second. In addition, the number of available nodes in clusters may differ greatly as an assignment of tasks to nodes in different clusters. WAN links between clusters are also shared by nodes in clusters, adding another level of complexity to the problem. Traditional algorithms that perform well inside a cluster, for example, the binomial algorithm, do not incorporate knowledge of these characteristics of a Grid system; so, they are not optimal for a Grid system. The problem of finding an optimal multicast solution in this environment is NP-hard and the complexity is high enough that an exhaustive search is infeasible. This dissertation solves this problem by modeling a Grid as a graph named a *grid graph* that represents a Grid system as a two-level network of clusters, each consisting of a different number of nodes. Then, a multicast operation is modeled as a multicast schedule which encapsulates the topology and the order of the operation. From the model, finding the optimal cost of multicast schedule can be done by constructing all possible instances of schedule, calculating the cost, and finding the optimal one. However, performing this operation is too time consuming to be practical except for using it as a theoretical lower bound.

This dissertation proposes a genetic-based algorithm named Genetic Algorithm-based Dynamic Tree (GADT) (Vorakosit and Uthayopas, 2003) that generates an efficient multicast tree for a Grid system. Experimental results show that GADT produces a multicast tree which has a communication performance close to the optimal multicast schedule. Moreover, the result is much faster than the traditional binomial algorithm. Due to the long running time of GADT, it can be used as an offline algorithm and a base-line for multicast algorithm research. For a runtime algorithm, this dissertation proposes a heuristic named *Longest Parallel Branch First (LPBF)* (Vorakosit and Uthayopas, 2004). This algorithm incorporates information about the two-level Grid topology, number of nodes in clusters, and link bandwidth to increase its efficiency. The efficiency of algorithm are measured by their total transmission time, which is the time required for all nodes participating in the operation to completely receive data from the originating node.

The proposed algorithms are implemented in a prototype MPI library named MPITH. It supports transparent communication among all nodes, even closed clusters, and contains an efficient collective communications algorithm. The library supports a subset of MPI functions that is large enough to develop a practical program (Vorakosit, 2003). This can lead to a deeper and more practical understanding of how to design and implement an efficient, scalable parallel runtime environment for a Grid system. MPTH is built on the top of Globus to provide Grid support. It uses Globus (Foster and Kesselman, 1997) runtime to execute a task in each cluster and synchronizes that task at the beginning of each execution. It contains a built-in MPI message forwarder that allows a forwarding of MPI messages across NAT gateways and LPBF multicast algorithm, which has been demonstrated experimentally to provide nearly optimal collective communications.

Objective

1. Explore new findings and algorithm that improve various aspect of building efficient MPI runtime library environment on a Grid system

   a. Propose a new global Grid routing strategy

   b. Propose new efficient multicast algorithms

2. Build a prototype MPI runtime library

3. Conduct performance evaluations using simulations and a real experiment to validate the proposed concepts

Research Scope

1. Propose a global Grid routing strategy

2. Propose efficient multicast algorithms on a Grid system

3. Develop a MPI compliant communication library equiped with proposed algorithms

4. Conduct performance evaluations

Background Theory

This chapter describes background theories required by this dissertation. This chapter begins with an introduction about parallel computer taxonomy. From a given parallel computer, there is a number of programming models used to harvest the potential of a parallel computer. Among those programming models, a message passing model is one that is widely used. It provides a uniform programming model ranging from supercomputers, clusters, to international Grid systems. In the past, there were many message passing libraries available. Each library had a different syntax and semantic. In order to enable portability of parallel applications, a standard was developed. The Message Passing Interface Standard (MPI) is one of the most widely used standards of a message passing library. MPI supports various data communication patterns. The classifications of MPI data communication is also provided in this chapter.

An MPI application must be run on a kind of a parallel computer. There are wide ranges of parallel computer architecture. Clusters are parallel computers that are distributed memory architecture. A cluster integrates complete personal computers, which is called *nodes*, with a high performance interconnection networkto form a high performance parallel computer machine with low cost. Grid technologies extend a concept of cluster into a wide area network. It is an emerging computing model that

provides the ability to perform high throughput computing by taking advantage of the many networked computers to model a virtual computer architecture that is able to distribute process execution across a parallel infrastructure.

This dissertation also uses genetic algorithms in order to solve a combinatorial optimization problem. The dissertation uses a PGApack library as the skeleton of genetic algorithm. The last two subsections in this chapter provide an introduction to genetic algorithm and PGApack library.

1. Parallel Computer Taxonomy

Traditionally, parallel computers are classified according to Flynn's taxonomy. Flynn's classification distinguished parallel computers according to the number of instruction streams and data operands being computed simultaneously.

Flynn's single-instruction single-data (SISD) model is the traditional sequential computer. A single program counter fetches instructions from memory; the instructions are executed on scalar operands. There is no parallelism in this model.

In the single-instruction multiple-data (SIMD) model there is again a single program counter fetching instructions from memory. However, now the operands of the instructions can be one of two types: either scalar or array. If the instruction calls for an execution involving only scalar operands, it is executed by the control processor (i.e., the central processing unit fetching instructions from memory). If, on the other hand, the instruction calls for execution using array operands, it is broadcast to the array of processing elements. The processing elements are separate computing devices that rely upon the control processor to determine the instructions they will execute.

In a multiple-instruction multiple-data (MIMD) computer, there exists multiple processors each of which has its own program counter. Processors execute independently of each other according to whatever instruction the program counter points to next. MIMD computers are usually further subdivided according to whether the processors share memory or each has its own memory.

In a shared-memory MIMD computer, both the program's instructions and the data to be shared exist within a single shared memory. Additionally, some data may be private to a processor and not be globally accessible by other processors. Processors execute asynchronously of each other. Communication and synchronization between processors are handled by having them each read or write a shared-memory location.

A distributed-memory MIMD computer consists of multiple nodes. A node consists of a processor, its own memory, a network interface, and sometimes a local disk. The program instructions and data reside in the node's memory. The nodes are connected via some type of network that allows them to communicate with each other. Parallelism is achieved by having each processor compute simultaneously on

the data in its own memory. Communication and synchronization are handled by the passing of messages (a destination node address and the local data to be sent) over the interconnection network.

2. Parallel Computation Models

There are three basic parallel computational models:

1. **Data parallelism**. Data parallelism that was made available to the programmer was in vector processors or SIMD machines. This architecture is motivated by the fact that many important scientific computations involved uniform calculations on every element of array or matrix. The key characteristic of the data parallel programming model is that operations can be performed on large data structure or array. In operation, the control processor broadcasts instructions to an array of data processing elements, which are connected as a grid or a matrix. The parallelism comes entirely from the data. The partitioning of data may be done by a compiler.

2. **Shared memory.** The shared memory multiprocessors provide better throughput on multiprogramming workloads, as well as to support parallel programs. Parallelism is explicitly specified by a programmer. Processors have access to all of a single shared address space by the load/store operations. There is a wide range of scale, from a few processors to hundreds. Now, there are many small scale shared-memory machines that are called *symmetric multiprocessors* or SMP.

3. **Message passing.** The parallel machines in this class are composed of a set of complete computer. The communications between processors are provided as explicit I/O operations; they are integrated at the I/O level by sending/receiving message rather than into the memory system. Message-passing models are implemented on a wide variety of hardware architectures.

3. Advantages of the Message-passing Model

Messaging passing model has many advantages over other models. This section gives some of its advantages.

1. Universality The message passing model can be applied to any architecture, ranging from the shared memory architecture to a Grid system. Nodes can be a set of complete computers or custom processors connected through any kind of network. The network can be any technology, for example, Ethernet, Fast Ethernet, Gigabit Ethernet, ATM, and so on.

2. Expressivity The message passing model expresses a parallel algorithm and also implies the control parallel. This helps the programmers to develop their algorithm more easily.

3. Ease of debugging The message passing model does not have a problem about a race condition that comes from memory reference. It requires explicit control over memory reference, data exchange and synchronization.

4. Performance The performance is the most compelling reason in parallel computing. The performance of the modern processor is based on cache and memory hierarchy. In message passing model, each processor controls its memory itself. So, when the number of nodes grows up, the number of cache grows, too. This yields to more locality of reference, so, the application that has more localities can runs faster.

4. The MPI Standard

In the past, each parallel machine vendors had their own syntax and semantic for their message passing library. So, the program developed for one vendor cannot be used with one from another vendor. In order to enable the portability of the application, the standard called *Message Passing Interface Standard* was developed. The standard was developed from a collection of widely used library, for example, PICL, PVM, PAMACS, p4, Chameleon, and ZipCode.

The standardization process began on April 29, 1992 in the workshop on Standards for Message Passing in a Distributed Environment. The draft MPI standard was presented at the Supercomputing 93 conference in November 1993. The goal of the Message Passing Interface was as follows:

1. Design an application programming interface.

2. Allow efficient communication.

3. Allow for implementations that can be used in a heterogeneous environment.

4. Allow convenient C and Fortran 77 bindings for the interface.

5. Assume a reliable communication interface.

6. Define an interface that is not too different from current practice, such as PVM, NX, Express, p4, etc.

7. Define an interface that can be implemented on many vendors' platforms.

8. Semantics of the interface should be language independent.

9. The interface should be designed to allow for thread-safety.

There are three major versions of the MPI standard. First MPI 1.1, finished in June 1995, described procedure specification, semantic terms, data types, language

binding, error handling, implementation issues, and the semantic of MPI operation. MPI 1.1 supports the following features:

1. Point-to-point communication
2. Collective operations
3. Process groups
4. Communication contexts
5. Process topologies
6. Bindings for Fortran 77 and C
7. Environmental management and inquiry
8. Profiling interface

MPI 1.2 and MPI-2 are both extensions to the MPI-1.1 standard. The MPI-1.2 provides clarifications and corrections to the MPI-1.0 and MPI-1.1 and defines addition to the MPI-1.2 standard. MPI-2 describes addition to MPI-1 standard and defines MPI-2. The new features of MPI-2 include: parallel I/O, interlanguage operation, process creation and management, and one-side communication

5. MPI Data Communication Classifications

MPI data communications can be categorized into two main classifications that are point-to-point, and collective communication. This section analyzes challenges in each type of communication.

5.1 Point-to-point Communications

In point-to-point (P2P) communications, there is a sender and a receiver involved in an operation. It can be divided into two modes; a blocking and a non-blocking. In blocking operations, a sender can continue an execution after a message is saved in a receiver's buffer; a receiver can continue an execution after a message is saved in a user's buffer. In non-blocking operations, both sender and receiver can continue an execution immediately after they have issued the command. They must explicitly check whether the operation is completed.

The challenges in a point-to-point communication are the presence of a firewall or a NAT gateway, and optimizing throughput of an operation. This dissertation proposes a solution to the first challenges, the presence of a firewall or a NAT gateway. It proposes an efficient and systematic methodology to discover routes between each pair of Grid nodes. By implementing a proposed methodology as a

built-in MPI message forwarder into MPI runtime library, this methodology provides a NAT transparency environment to MPI applications.

Optimizing throughput of an operation is the second challenge in point-to-point communications. There is a number of proposed method that addressed this challenge, for example, tuning TCP parameter, optimizing MPI protocol, reducing message copying and so on. However, those methods work fine within a low-latency high-bandwidth network that is an intracluster network. In a Grid system, a network is high-latency low-bandwidth; the network transmission time dominates the advantages from those methods. Moreover, a Grid is a dynamic system. The parameter tuned for a time period may not be used later. In a Grid system, possible solutions are routing a message using multipath and routing a message through other MPI processes.

5.2 Collective Communications

In collective communications (CC), there is more than one process involved in the operation. The operation can be either nonpersonalized or personalized.

Nonpersonalized collective communications (NPCC) are operations that process's rank does not affect the result of an operation, for example, MPI_Bcast, and MPI_Reduce. An NPCC has three modes: one-to-all (MPI_Bcast), all-to-one (MPI_Reduce), and all-to-all (MPI_Allreduce, MPI_All_to_all). This dissertation proposes two solutions to this 1-to-all and all-to-1 problem. The one is based on genetic algorithm and the other one is a heuristic algorithm. The heuristic algorithm named LPBF is for MPI runtime library. Because all-to-1 is a reverse of one-to-all, LPBF can also be used to generate a multicast schedule for all-to-one operation mode.

Personalized collective communications (PCC) are operations that process's ranks affect the result of an operation. A PCC, like NPCC, has three modes: one-to-all (MPI_Scatter), all-to-one (MPI_Gather), and all-to-all (MPI_Alltoall). PCC differs from NPCC because a message size of PCC is changed when a message is forwarded through an MPI process. In the case of NPCC, data sent to each process is the same. However, in case of PCC, when a process receives data, it removes its own data and forwards the remaining packet to its children. So, the data size is changed all the time. This adds a complexity to the problem. This operation is not in the scope of the dissertation. Currently, the prototype implementation uses a flat tree algorithm for intracluster PCC. For an intercluster PCC, the cluster of head node of each cluster is formed, and the message is sent to these head node using a flat tree algorithm. After that, the head nodes use a flat tree to perform their intracluster PCC.

6. Clusters

A cluster is a collection of complete computers, called *node* or *host* that is physically interconnected by a high-performance interconnection network. Typically, each computer node may be a SMP server, a workstation or a personal computer. More importantly, all cluster nodes must be able to work together cooperatively as a

single, integrated computing resource; in addition to fill the conventional role of using each node by interactive users individually. Figure 1 shows a typical architecture of cluster.

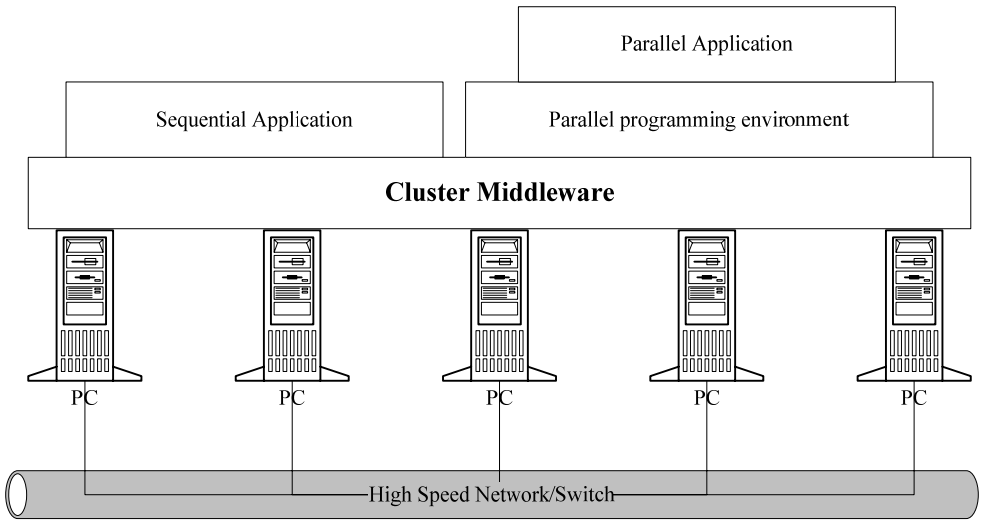

Figure 1 A cluster architecture

7. Grid Systems

Grid computing is distributed computing at the next evolutionary level. The goal is to create the illusion of a simple yet large and powerful self-management virtual computer out of a large collection of connected heterogeneous systems sharing various combinations of resources. The easiest use of grid computing is to run an existing application on a different machine. The benefits of grid computing are exploiting underutilized resources, parallel CPU capacity, virtual resources and virtual organizations for collaboration and resource balancing.

Grid requires many software components. Some of key components are:

1. Management component. There are three responsibilities of the management component. First, it keeps track of the resources available to a Grid and determines which users are members of a Grid. Second, it determines both the capacities of the nodes on a Grid and their current utilization rate at any given time. Third, advanced grid management software can automatically manage many aspects of a Grid. This is known as autonomic computing.

2. Donor software. Each machine contributing resources typically needs to enroll as a member of a Grid and install some software that manages a Grid's use of its resources. Some identification and authentication procedure must be performed before a machine can join a Grid. A Certificate Authority can be used to establish the identity of the donor machine as well as the users of the Grid itself.

3. Submission software. This is a portal to submit jobs to a Grid and initiate Grid queries. This software is usually installed on the user's desktop or workstation.

4. Schedulers. This software is located on a machine which to run a Grid job that has been submitted by a user.

5. Observation, management, and measurement. Usually, the donor software will include some tools that measure a current load and an activity on a given machine using either operating system facilities or by direct measurement. The measurement information can be saved for accounting purposes, or to form the basis for a Grid resource brokering, or to manage priorities more fairly.

Nowadays, the Globus Toolkit is a set of tools useful for building a Grid. Its strength is a good security model, with a provision for hierarchically collecting data about a Grid, as well as the basic facilities for implementing a Grid.

The year 2002, Open Grid Services Architecture (OGSA) (Foster *et al.*, 2002) is a true community standard with multiple implementations. OGSA firmly aligns a Grid computing with broad industry initiatives in a service-oriented architecture and Web services. The three principal elements of OGSA are:

1. Open Grid Services Infrastructure – provides standard interfaces and semantics. In combination with a web services definition language (WSDL), OGSI defines mechanisms for creating, naming, managing lifetime, monitoring, grouping, and exchanging information among entities called Grid services.

2. OGSA services – provides services in OGSA

3. OGSA schemas – interface definition based WSDL

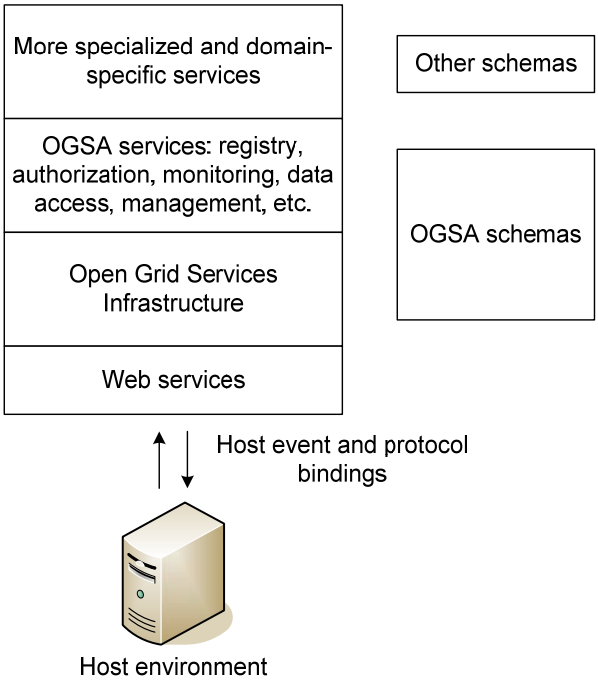

Figure 2 The core elements of the OGSA

One example of a Grid is ThaiGRID (Varavidthaya and Uthayopas, 2000). ThaiGRID is a collaboration network between universities and research institutes. Currently, tgcc.cpe.ku.ac.th is a portal for ThaiGRID. It provides a job submission facility. SQMS-G (Phatanapherom and Uthapoyas, 2002) is a grid scheduler for ThaiGRID.

8. Genetic Algorithms

Genetic algorithms (GA) (Goldberg, 1989) is searching algorithm developed by John Holland, his colleagues, and his student at the University of Michigan. They are based on the mechanism of natural selection and natural genetics. They combine survival model of the fittest among strings with a structured that randomized information exchange to form a search algorithm. In every generation, a new set of artificial creatures is created using string and fittest of the old; and occasional new part is tried for good measure.

Genetics algorithms' problem is encoded as a string of specific data type, usually integer or binary. These strings represent a deoxyribonucleic acid (DNA) of an artificial creature. From the theory of selection of Darwin, the ability of a creature to reproduce is the fitness of that creature. In GA, the fitness value of each string is determined by a fitness function or an objective function. Users must develop their fitness function themselves.

A mechanism of a basic genetic algorithm is very simple. It is a copying string and swapping partial strings. There are three operators that take the population and

generate successive populations that should improve over time. These operators are reproduction, crossover and mutation operator.

The first operator, a reproduction operator is a process in which individual strings are copied according to their fitness values. The strings that have more fitness value means that they have a higher probability to contribute one or more offspring in the next generation. The exact replica of the sprint is made. Then, this string enters into a mating pool or a next genetic operator.

The second operator is a crossover operator. After the reproduction, the crossover operator is applied. This operator has two steps. First, members of a new string in the mating pool are mated randomly. Second, the value $1 < k < string\ length$ is generated randomly. Two new strings are created by swapping all elements between position $k+1$ and string length inclusively.

The last operator is a mutation operator. The mutation is a phenomena in which the element string is changed immediately. The mutation is needed because even though reproduction and crossover operators effectively search and recombine notations, occasionally they may lose some useful genetic materials. In artificial genetic systems, the mutation operator protects this unrecoverable loss. In the simple GA, mutation is the occasional random alteration of the value of a string position. If the data is coded as a binary value, mutation means changing a 1 to a 0 and vice versa. The frequency of mutation to obtain good results in an empirical genetic algorithm is on the order of one mutation per thousand bits or position transfers. Because the mutation operator is taken in very low probability, the mutation is considered as a secondary mechanism of genetic algorithm adaptation.

In some sense, strings are not interested as strings alone. A schema is a similarity template describing a subset of strings with similarities at certain string positions. For example, without loss of generality, string is a binary alphabet *{0, 1}*. The search space has three bits. This can be represented as a simple cube with the string 000 at the origin. The corners in this cube are numbered by bit strings and all adjacent corners are labeled by bit strings that differ by exactly 1-bit, as shown in Figure 3.

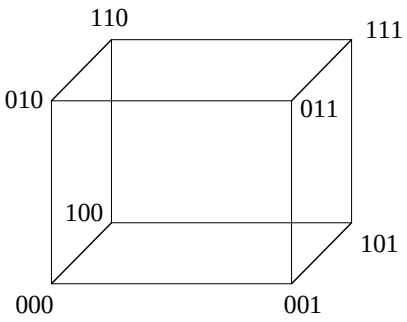

Figure 3 A 3-dimensional cube for binary DNA string of length 3

The front plane of the cube contains all the points that begin with 0. If "*" is used as a "don't care" or wild card match symbol, then this plane can be also represented by the special string 0**. Strings that contain * can be referred to as schemata; each schema corresponds to a hyperplane in the search space. The *order* of a hyperplane refers to the number of actual bit value that appears in its schema. Thus, 1** is order one while 1**1****0** would be of order three. With this extended alphabet, strings can be created over the ternary alphabet {0, 1,*}. The idea of a schema gives a powerful and compact way to talk about all the well-defined similarities among finite-length strings over a finite alphabet. Every binary encoding is a member of $2^L - 1$ different hyperplanes, where L is the length of the binary encoding. In other words, there are L positions in the bit string and each position can be either the bit value contained in the string or the "*" symbol. In general, for alphabets of cardinality k, there are $(k+1)^L$ schemata.

Establishing that each string is a member of 2^L-1 hyperplane partitions doesn't provide very much information if each point in the search space is examined in isolation. This is why the notation of a population based search is critical to genetic algorithms. A population of sample points provides information about numerous hyperplanes; furthermore, low order hyperplanes should be sampled by numerous points in the population. The key part of the genetic algorithms' intrinsic or implicit parallelism is derived from the fact that many hyperplanes are sampled when a population of strings is evaluated; in fact, it can be argued that far more hyperplanes are sampled than the number of strings contained in the population. Many different hyperplanes are evaluated in an implicitly parallel fashion each time a single string is evaluated, but it is the cumulative effect of evaluating a population of points that provides statistical information about any particular subset of hyperplanes.

Implicit parallelism implies that many hyperplane competitions are simultaneously solved in parallel. The theory suggests that through the process of reproduction and recombination, the schemata of competing hyperplanes increases or decreases their representation of population according to the relative fitness of the strings that lie in those hyperplane partitions.

Keep in mind that a symbol * is only a metasymbol (a symbol about other symbols); it is never explicitly processed by the genetic algorithm. It is simply a notational device that allows description of all possible similarities among strings of a particular length and alphabet.

9. PGAPack

PGAPack (Levine, 1996) is a parallel genetic algorithm library that is intended to provide most capabilities desired in a genetic algorithm package, in an integrated, seamless, and portable manner. Key features of PGAPack are as follows:

1. Ability to be called from Fortran or C.

2. Executable on uniprocessors, multiprocessors, multicomputers, and workstation networks.

3. Binary-, integer-, real-, and character-valued native data types.

4. Object-oriented data structure neutral design.

5. Parameterized population replacement

6. Multiple choices for selection, crossover, and mutation operations

7. Easy integration of hill-climbing heuristics.

8. Easy-to-use interface for novice and application users.

9. Multiple levels of access for expert users.

10. Full extensibility to support custom operations and new data types.

11. Extensive debugging facilities

12. Large set of example problems.

A parallel version of PGAPack was built on the top of an MPI library. PGAPack 1.0 supports sequential and parallel implementations of the single population global model (GM). The parallel implementation uses a master/slave algorithm in which one process, the master, executes all steps of the genetic algorithm except function evaluations. The function evaluations are executed by slave processes. However, in the special case of exactly two processes, the master executes function evaluations as well.

The parallel implementation of the GM will produce the same result as the sequential implementation, usually faster. However, the parallel implementation varies with the number of processes. In general, the speedup obtained will vary with the amount of computation associated with a function evaluation and the computational overhead of distributing and collecting information to and from the slave processes.

The speedup that can be achieved with the master/slave model is limited by the number of function evaluations that can be executed in parallel. This number depends on a population size and the number of new strings created in each generation. For example, if the population size is 100 and 100 new strings are created each GA generation, and then up to 100 processors can be put to effective use to run the slave processes. However, with the default rule of replacing only 10% of the population each GA generation, only 10 processors can be used effectively.

LITERATURE REVIEW

This dissertation proposes a model of communication in a Grid system, an implementation of communication library, and evaluating communication model. This chapter presents various researches related to this dissertation.

Wide Area Collective Communication Algorithms

The problem of finding a good multicast algorithm in a cluster and a Grid environment has been addressed by several researchers. These researches can be categorized into five categories: parameter, topology, schedule, heuristic, and application.

The parameter-based optimization tends to precisely model a multicast operation. Most models are based on LogP (Culler *et al.*, 1993). LogP consists of four parameters: *L* (latency), *o* (overhead), *g* (gap), and *P* (the number of processor). Karp *et al.* (1993) proposed optimal multicast and summation algorithms based on LogP. They addressed six fundamental communication problems: single-item multicast, k-item multicast, continuous multicast, all-to-all multicast, combining-multicast, and summing. For a single-item multicast, which is focused in this dissertation, a binomial algorithm was used. Karp proved the optimality of this algorithm under the LogP model; however, an optimality is only valid in a network where the inter-node bandwidths can be assumed to be equal which is only valid within clusters. Moreover, LogP model is suitable only for short messages. Alexandrov *et al.* (1995) proposed LogGP as an extension of LogP for long messages. He added a new parameter, *G*, that is a gap per byte for a long message. Iannello (Iannello, 1997) used the model to implement reduce-scatter operation. However, the above models were limited in that bandwidth between each link must be equal.

For multicast operation in a homogeneous network, a binomial algorithm is widely used because it optimizes the completion time; however, when the link transmission times are not equal, a binomial algorithm may be very inefficient. Banikazemi *et al.* (1998) proposed a communication model for heterogeneous networks of workstations. Their model incorporates a send-overhead, a transmission time, and a receive-overhead to compute a transmission time of a link. A multicast tree shape is then selected to minimize the completion time using these parameters. His model is used to optimize a multicast operation; he does not provide a multicast algorithm for his model. Bernaschi and Iannello (1998) proposed a multicast algorithm based on an α-tree. The value α must be given by the user before the tree is generated. MagPIe (Kielmann *et al.*, 1999) used this model to implement an add-on library for MPICH. Kielmann *et al.* (2001) proposed a model named *Parameterized LogP* and multicast messages are divided into segments. An objective was to find a multicast tree and a segment size that minimizes the completion time. This model is used in the later version of MagPIe. Le and Rejeb (Le and Rejeb, 2006) extended LogGP to support a Grid system. They introduced a new parameter, an *effective latency*, to predict a behavior of a point-to-point message transmission. However, this still supports only a point-to-point communication. This optimization category

requires system parameters in detail. Parameters can be changed when a Grid system changes. Algorithms proposed in this dissertation minimize the number of parameters. So, they can be used in any Grid configuration.

The topology-based optimization studies a structure of multicast tree generated from a multicast algorithm. Karonis *et al.* (2002) proposed a modification to MPICH to support hierarchical networks. Their method optimized a broadcast operation by computing a broadcast schedule at each level of the hierarchy. Beaumont *et al.* (2005) proposed broadcast trees for heterogeneous platforms. The model is based on a spanning tree that describes how to generate a broadcast tree. However, this optimization category did not address an ordering of transmission between each process, which could significantly improve an overall performance of multicast algorithm.

The schedule-based optimization models not only a multicast topology but also orders of transmission within the operation. There are several schedule-based optimizations proposed. Vadhiyar *et al.* (2000) proposed an implementation-based performance tuning by forcing all non-root nodes send start-acknowledge package to a root in order to allow the root selecting a communication pattern on the fly. Bhat *et al.* (2003) proposed an algorithm called Earliest Completing Edge First (ECEF) for selecting a communication schedule. ECEF constructs a tree iteratively: at each step ECEF adds the node which minimizes the completion time for each branch in the tree. ECEF is a kind of greedy algorithm: the sender is a set of nodes that already has data and the receiver is a set of nodes that needs the data; at each iteration, the fastest edge from sender to receiver is used to choose a node to add to the sender set. The process continues until the receiver set is empty. However, ECEF algorithm can be improved by taking into account an overlapping of communication. Wu *et al.* (2004) proposed six heuristic models for multiple multicast on heterogeneous network of workstations in non-blocking transmission operation. They showed that Earliest-Completion-First algorithm provides the best performance among their proposed model. This dissertation extends the schedule-based optimization by maximizing communication overlapping in order to reduce an overall total transmission time.

The heuristic-based optimization optimizes multicast operations using some heuristic algorithms. For example, Faraj and Yuan (2005) proposed automatically generating topology-specific routines from four parameters: communication time, sequentialization overhead, synchronization overhead and contention overhead. In this model, there are algorithm repositories for each type of operation. For instance, possible algorithms for a broadcast operation are sequential, binomial, binary, ring algorithm and so on. The algorithm is selected from the cost model of those four parameters. However, the model supported only Ethernet switched clusters. Algorithms proposed in this dissertation provide a general algorithm for a Grid system.

Finally, the application-based optimization is an optimization at application level. Bal *et al.* (1998) measured the performance of eight parallel programs in WAN. He proposed that high performance can be obtained if the programs are optimized to

take the multilevel network structure into account to reduce intercluster traffic and hide intercluster latency. However, the optimization method is fixed to a specific application. This dissertation provides a general algorithm that is suitable for any type of MPI application.

MPI Implementations

As mentioned above, MPI is one of the well established parallel programming models for distributed computing. Many MPI implementations have been developed; of these, the most popular is MPICH (Gropp *et al.*, 1996), developed at Argonne National Laboratory. MPICH provides a portable, high performance MPI implementation for clusters, and an extensible *Abstract Device Interface* (*ADI*) that allows developers to extend MPICH to address specific issues. A newer implementation, MPICH2 (Gropp, 2005), which supports the full MPI-1.0 and 2.0 protocols, provides improved optimization of communications, and incorporates new MPI-2 features, such as remote memory access. LAM (Burns *et al.*, 1994; Squyres *et al.*, 2003), is another MPI implementation that provides many additional features such as OpenPBS integration, beta grid support, high-speed interconnection support, check-point and restart, fast start-up. Open MPI (Gabriel *et al.*, 2004), a recent collaboration between several universities and Los Alamos National Laboratory, aims to integrate technologies from several MPI projects into a fast, efficient, MPI-2 compliant implementation. Open MPI currently focuses on fault-tolerance, heterogeneity, and performance rather than providing seamless Grid support.

Several Grid-enabled MPI libraries have been developed: MPICH-G2 (Karonis *et al.*, 2003) is an extension of MPICH that provides authentication, scheduling, and resource management required for a Grid environment, using Globus as a Grid-interface subsystem.

MPICH-G2 optimizes collective communications on a Grid using a multi-level topology approach. MagPIe, another MPI library that extends MPICH, attempts to optimize communication algorithms for wide area networks using the *Parameterized LogP*. Other projects attempt to optimize MPI communication by tuning protocol parameters. GridMPI (Matsuda *et al.*, 2005) incorporates an algorithm to build a latency aware communication library by modifying Linux TCP stack and tuning TCP parameters. Karmal *et al.* (Kamal *et al.*, 2005) modified LAM to use SCTP (Stewart and Xie, 2001) instead of TCP for MPI communication. SCTP is a new protocol that is a message oriented and provides connection management, congestion and a flow control mechanism. SCTP can define subflows inside a single connection that eliminates the head-of-line blocking that can occur in TCP-based MPI middleware. The MPITH project has also proposed topology-aware communications using LPBF, might seem boastful that reduces latency by scheduling transmissions along the longest branches first.

The grid-enabled MPI libraries cited above do not address the problem of utilizing nodes in closed clusters, or a presence of NAT gateways. One solution to this is to incorporate a routing and forwarding mechanism into an MPI runtime library,

with cluster gateways acting as forwarders. The forwarders can be implemented in mainly two approaches: operating system, and application level. Das *et al.* (Das *et al.*, 2005) uses *Realm Specific IP (RSIP)* (Borella *et al.*, 2001) as a replacement for NAT. RSIP avoids address translation by leasing ports of a public IP to nodes inside a cluster. However, this model requires a system administrator privilege to configure. Moreover, a development of kernel module for RSIP is a high complexity task.

An application-level forwarder, on the other hand, has several advantages: ease of upgrade, simple installation, and configuration. An application-level forwarder can be either shared or nonshared. A shared forwarder typically runs as a system daemon and provides forwarding service to all MPI applications. PACX-MPI (Brune *et al.*, 1999) uses this approach: an intracluster communication takes place directly among tasks, while inter-cluster communication is done through communication nodes called MPI-servers. An MPI-server compresses data and transfers it via TCP/IP to another communication node, called a PACX-server, which decompresses the data and sends it to a target node. This is a viable solution, but it also has drawbacks. First, MPI- and PACX-servers have to be installed and maintained on each cluster, complicating the administration. Second, the MPI-server will run and consume resources continuously even if no MPI programs are being used. Third, the presence of a separate daemon complicates the runtime system and adds difficulties to enforcing a Grid security model.

An alternative to a shared forwarder, a non-shared forwarder is built into an MPI runtime library. Each instance of application has its own forwarder. This also promotes strict enforcement of a grid security model. A non-shared forwarder may also execute the user code along with the forwarding routine, called a *runnable* forwarder, or execute only the forwarding routine called a *nonrunable* forwarder. MPICH/MADIII (Aumage and Mercier, 2003) uses the non-shared runnable approach. It implements an MPICH device on top of the Madeleine III communication library. MPICH/MADIII is designed as a complementary tool for MPICH-G2, However, interfacing between MPICH/MADIII and MPICH-G2 complicates the runtime. In contrast, the proposed approach integrates both grid-level and cluster-level message routing into a single MPI implementation. Hence, it provides full transparency and global optimization between a Grid and a cluster level. Table 1 illustrates a features comparison between MPI implementations.

Table 1 Features comparison between MPI implementations

Features / Implementation	Cluster	Grid	Topology-aware	Optimization	NAT/share	NAT/nonshare
MPICH/MPICH2	×					
LAM	×					
Open MPI	×					
MPICH-G2	×	×	×			
MagPIe	×	×	×			
GridMPI	×	×		×		
Karmal *et al.* (2005)	×	×		×		
Das *et al.* (2005)	×	×			×	
PACX-MPI	×	×			×	
MPICH/MADIII	×					×
MPITH	×	×	×			×

The experimental MPITH library described here uses a nonshared, runnable application mode forwarder, as this is simpler to implement and configure. An implementation of nonrunnable forwarder is for future work.

MATERIALS AND METHODS

Materials

This dissertation uses a cluster to develop a prototype MPI runtime library. The cluster used in this research is the Magi cluster. It is a four-node cluster, Athlon XP 2500+ with 512 MB RAM. All nodes are connected through Gigabit Ethernet switch. Figure 4 shows the diagram of Magi cluster.

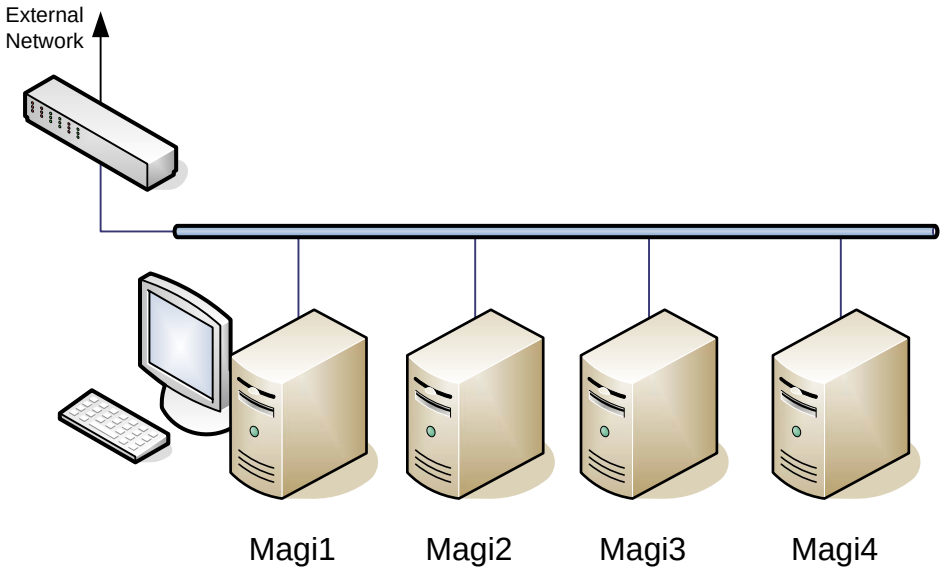

Figure 4 Magi cluster

The hardware and software requirement for this research is as follows:

1. Hardware requirement
    a. Magi cluster
    b. PC for developing a library. Athlon XP 1800+ with 256 MB RAM
2. Software requirement
    a. GNU/Linux operating system
    b. GNU C++ Compiler (bundle with OS)
    c. Concurrent Version System
    d. Utilities software such as autoconf, automake, make

Methods

This section describes an overview of the methods used in this dissertation:

1. Model a Grid system. A Grid system is modeled using graphs called a *Virtual cluster* model. There are two graphs in VC model. The first graph is called a Grid graph. It represents a network bandwidth between Grid sites, which are usually clusters. This graph is used when modeling a multicast operation and calculating a total transmission time of a multicast operation. The second graph, a *connectivity graph*, is a representation of logical network connection between nodes in a Grid system. This graph illustrates a connection initialization between nodes. It is used in a global Grid routing algorithm. This graph is simplified to a *virtual cluster* graph. The route is computed from the virtual graph in order to reduce discovery time.

2. Propose a global Grid routing strategy. From the Grid model described in 1, a routing table is calculated from the virtual cluster graph. This routing table describes a next hop virtual cluster from source to destination in term of virtual cluster. At runtime, an MPI process, which is responsible for forwarding an MPI message, is identified by *MPI process identifier (MPID)* that equals to a rank in MPI_COMM_WORLD. There must be auxiliary functions that map between a virtual cluster to an MPID.

3. Model an MPI multicast operation. The multicast operation is a communication among a number of processes. This dissertation focuses on one-to-all nonpersonalized collective communication, which is MPI_Bcast function in the MPI semantic. For a given multicast operation, there is a number of methodologies to achieve the operation. Each instance is called a *multicast schedule*. A multicast schedule describes a structure of data transmission and dependencies between each transmission in a multicast schedule. Both structure and dependencies impact an overall total transmission time of a multicast schedule. The model assumes a fully connected network between source and destination. So, for a given set of nodes in a Grid, there are many multicast schedules for a given multicast operation. Finding an optimal multicast schedule using an exhaustive search is $O(n^{2n})$. It is impossible to find an optimal multicast schedule using an exhaustive search.

4. Propose a new multicast algorithm based on genetic algorithms called Genetic Algorithm-based Dynamic Tree (GADT). In GADT, a multicast schedule is represented in chromosome. A total transmission time is a fitness value of the chromosome. An advantage of GADT is a near optimal solution can be found in an acceptable time.

5. Propose a new heuristic algorithm called Longest Parallel First (LPBF) algorithm. Even though GADT can create a near optimal solution in an acceptable time, it is too high to use in MPI library. LPBF is a heuristic algorithm that can calculate a good multicast schedule. LPBF uses an efficient structure of data transmission in conjunction with an intelligent message scheduling. By scheduling a

long branch in multicast tree earlier, subsequent data transmissions will overlap with a longer branch. This is a result of a shorter total transmission time.

6. Implement a simulator in order to evaluate proposed multicast algorithms. A small simulator named TreeSim is developed. This simulator is designed to support bandwidth sharing between clusters. It takes a Grid configuration and a multicast schedule as input and calculates a total transmission time. This simulator must be small and calculates output within a short period of time. These are important properties because the simulator is used as a fitness value calculator in GADT. So, it evaluates a number of multicast schedules because GADT performs searching on a large number of multicast schedules and iteratives until a near optimal schedule is found.

7. Implement a prototype MPI library that equips with a global Grid routing strategy and LPBF algorithm. A prototype MPI library named MPITH is developed as a proof of concept MPI library. MPITH supports basic MPI functions that are large enough for developing a practical MPI application. MPITH provides NAT transparency by implementing a proposed global Grid routing strategy. An MPI message forwarder is implemented as an I/O thread in MPITH process. It uses Globus/Duroc for intercluster task execution and synchronization. MPITH supports a compile-time pluggable collective communication algorithm. MPITH contains an implementation of LPBF for a Grid-level multicast operation.

The rest of this chapter describes each method in detail.

A Virtual Cluster Model

This section gives the model about a Grid system. A Grid system is modelled as graphs. There are two types of graph. The first one is a *Grid graph* that is used to describe bandwidth between clusters. A Grid graph is used in a multicast algorithm. The second one is a *virtual cluster* graph that is used to model connection between nodes. A virtual cluster is a group of nodes that can initiate a connection to all other nodes in the group. The virtual graph is created from a set of virtual clusters and used in a routing discovery algorithm.

First, a *cluster* is a group of computers or *nodes* connected together through an interconnection network. A designated *front-end node* acts as a management point for the cluster. In this dissertation, the term *physical cluster* refers to this type of cluster. There are two types of physical clusters: closed and open. In a closed physical cluster, all compute nodes are placed behind the front-end node. The front-end or the *gateway node* provides a NAT facility for other nodes. In an open physical cluster, all compute nodes are directly addressable from an outside network. A *Grid* is an interconnected set of these clusters. Figure 5 shows a Grid consisting of three clusters: Mercury, Venus, and Jupiter. The Venus and Jupiter clusters are closed physical clusters while Mercury is an open physical cluster. Venus1, Jupiter1, and Mercury1 are the front-end nodes of their respective clusters.

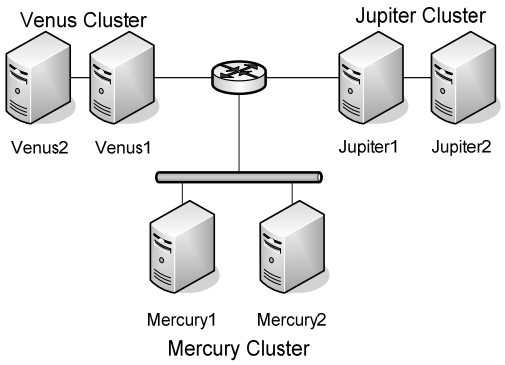

Figure 5 An example of a Grid environment

A Grid system is formally modeled as a weighted graph called a *Grid graph* as defined in Definition 1. The grid graph can be represented as a *bandwidth matrix* as defined in Definition 2.

Definition 1 The Grid graph

The *Grid graph* is a directed weighted graph $G = (V, E)$ where V is a set of clusters and E is a set of edges. The associated weighted function $w: E \times E \rightarrow R$ represents bandwidth between each cluster. Figure 6 shows a Grid graph of Figure 5.

Definition 2 The Bandwidth matrix

The *bandwidth matrix (B)* is an $n \times n$ matrix, where n is the number of clusters in a Grid. Each element b_{ij} is a link bandwidth from cluster i to j if $i \neq j$; an internal bandwidth of a otherwise. Figure 7 shows the bandwidth matrix for Figure 6.

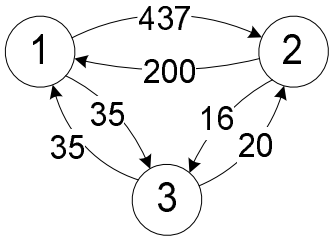

Figure 6 A Grid graph derived from Figure 5

$$\begin{bmatrix} 1000 & 437 & 35 \\ 200 & 1000 & 16 \\ 35 & 20 & 1000 \end{bmatrix}$$

Figure 7 A bandwidth matrix for Figure 6

The Grid graph represents a Grid in a cluster-level. The graph can be expanded to represent a Grid in a node-level called a *connectivity graph (CG)*. It represents connection initializations between nodes. Definition 3 defines the term formally.

Definition 3 The Connectivity Graph (CG)

The *connectivity graph (CG)* is a directed graph $G_c = (V_c,E_c)$ where V_c is the set of nodes in a grid and E_c is the set of edges. An edge $(u, v) \in E_c$ if and only if node u can initiate a connection to node v. Figure 8 shows the CG derived from Figure 5.

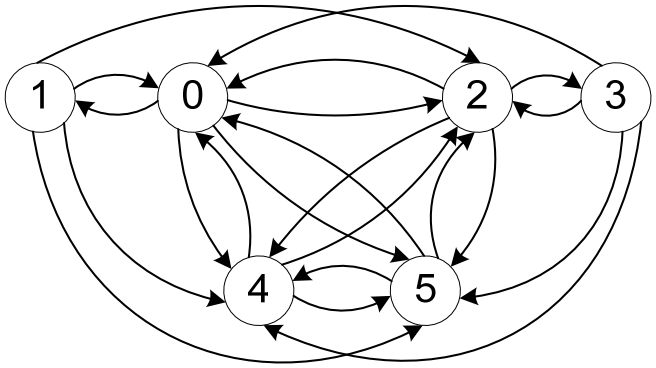

Figure 8 Connectivity graph of Figure 5

In a NAT environment, a node behind a gateway can initiate a connection to outside nodes but not vice versa since it cannot identify the destination address. For example, Mercury2 cannot initiate a connection to Venus2 if Venus2 is in a NAT environment behind Venus1. Although a connectivity graph provides useful information, this representation of network topology is too complex for our purposes. A connection between two nodes is useful only if a connection initiation can be done bi-directionally. Based on this observation, a connectivity graph can be reduced to a simpler *direct connectivity graph (DCG)*, as in Definition 2.

Definition 4 The direct connectivity graph (DCG)

The *direct connectivity graph (DCG)* is an undirected graph $G_d = (V_d\ E_d)$ where $V_d = V_c\ E_d$ is the set of edges. An edge $e \in E_d$ if and only if $(u,v) \in E_c$ and $(v,u) \in E_c$.

This undirected graph can be constructed from a CG by the following simple algorithm:-

Algorithm 1 A DCG construction

```
Input: CG Gc = (Vc,Ec)
Output: DCG Gd = (Vd, Ed)
Vd = V
Ed = ∅
for each (u,v) in Ec
     if (v,u) ∈ Ec
```

$E_d = E_d \cup \{ (u,v) \}$

Next

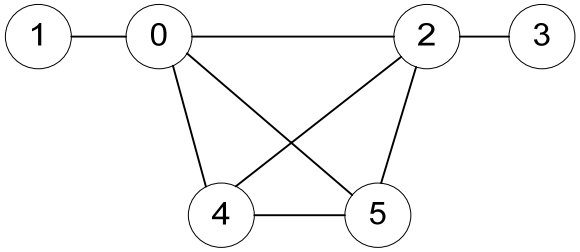

Figure 9 A DCG derived from Figure 8

Figure 9 shows the DCG resulting from Figure 8. With this concept, the task needed is finding a systematic way to map MPI tasks onto a Grid and provide routes that ultimately organize the tasks into a DCG. Hence, all tasks will have a way to communicate with each other regardless of any NAT encountered.

The gateway nodes are important because a process in this node not only performs computation, but also helps route messages to and from nodes inside the cluster as well. So, there must be at least one process mapped onto this node. A *gateway node* is defined in Definition 5.

Definition 5 The gateway node

The *gateway node* of a cluster is a vertex $v \in V_d$ such that the reduced set $V_d' = V_d - \{v\}$ causes $G' = (V_d', E_d')$ to be divided into two connected components.

The mapping proposed in this dissertation is based on the concept of building a set of *virtual clusters* on a Grid of physical clusters. A *virtual cluster (VC)* is a set of nodes such that every node can initiate a bi-directional connection to every node in the cluster. Let G_d be a DCG of a grid. Definition 6 defines a virtual cluster and its vertices.

Definition 6 The virtual cluster

The *virtual cluster*, denoted by *VC*, is a maximal subgraph of G_d such that given $u,v \in V_d$ then $(u,v) \in E_d$ and $(v,u) \in E_d$

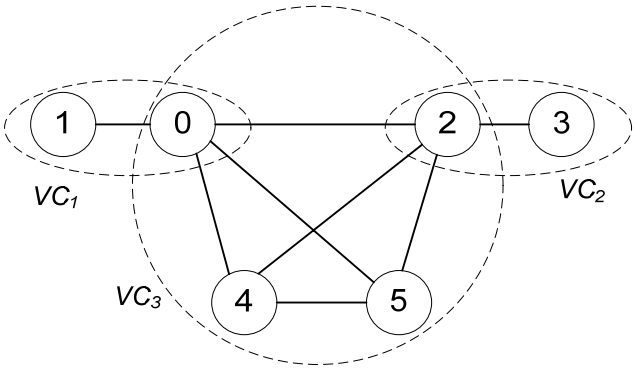

Figure 10 ***Virtual clusters*** **in** G_d

Figure 10 illustrates three *VC*s in *DCG*. From a given DCG, there are many virtual clusters. Those virtual clusters can be categorized into *level* by the maximum path length within a virtual cluster. The level of virtual cluster is defined in Definition 7 and Definition 8.

Definition 7 **The level one virtual cluster**

The *level one virtual cluster*, denoted by *vc[1]*, is a maximal subgraph of DCG such that the shortest path length $u \rightarrow v$ is 1 for all nodes *u, v* in the subgraph.

Definition 8 **The level *i* virtual cluster**

The *level i virtual cluster*, denoted by *vc[i]*, is a subgraph of DCG such that the shortest path length $u \rightarrow v$ is at most *i* for all nodes *u, v* in the subgraph.

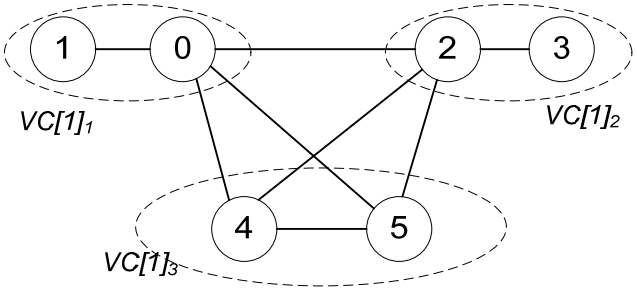

Figure 11 ***Vc[1]*** **in** G_d

Figure 11 illustrates three *vc[1]*s in DCG. The group of *vc[1]* in DCG is called a *vc[1] set*. Note that *vc[1]* is a strongly connected component in DCG and an edge $(u,v) \in E_d$ if and only if *u* and *v* are in the same *vc[1]*. As for *vc[1]*, there are more than one virtual clusters for each level of them in DCG. These *vc[i]* can be grouped into a *vc[i] set*. Figure 12 and Figure 13 shows that *vc[1]* can be merged into a higher level. The gateway vertex is important because in MPI process mapping, we always map one process onto this node. This process not only performs computation, but also routes messages to and from nodes inside the cluster as well.

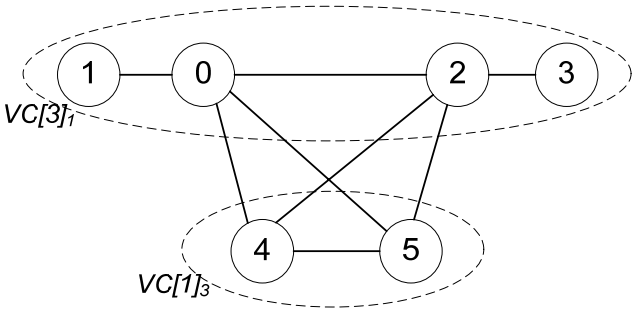

Figure 12 Intermediate step of VC merging

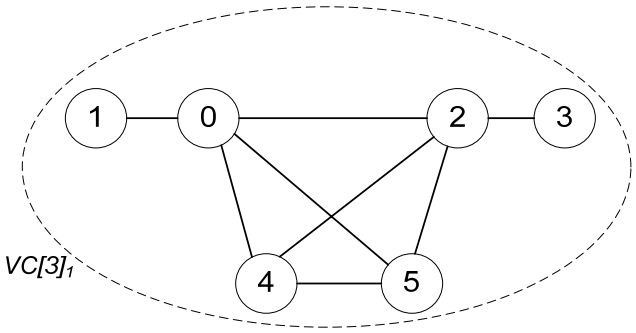

Figure 13 Final VC merging

The maximum level of virtual clusters in DCG is equal to the longest path length in it. Using this approach, a mapping of MPI tasks onto physical clusters can be performed by iteratively merging lower level virtual clusters into a higher level virtual cluster. Hence, routing can easily be derived by sending messages from a top level virtual cluster in DCG down the hierarchy of virtual clusters. Another important property is that a *vc[i]* is always a connected component. The proof of this is as follows.

Lemma 1 A *vc[1]* is a connected graph.

Proof: This is true by the definition of a virtual cluster. ■

Lemma 2 A *vc[i+1]* is a connected graph.

Proof: Let u, $w_1 \in vc[i]_1$ and v, $w_2 \in vc[i]_2$. Moreover, let w_1 and w_2 be gateway nodes of $vc[i]_1$ and $vc[i]_2$, respectively. From the DCG, there exists an edge (w_1, w_2). Hence, there is a path $u \rightarrow v$ is $u \rightarrow (w_1, w_2) \rightarrow v$. Hence, *vc[i+1]* that is constructed by merging of $vc[i]_1$ and $vc[i]_2$ is a connected graph. Hence, there always exists a route from any node in $vc[i]_1$ to $vc[i]_2$. ■

Theorem 1 A *vc[j]* is a connected graph at any level *j*.

Proof: The proof is by induction. First, for $j = 1$, *vc[1]* is a connected graph by *Lemma 1*. Then, for the inductive step, suppose all *vc[i]* are connected graphs, *vc[i+1]* that is built from the merging of two or more *vc[i]* is a connected graph

according to Lemma 2. Therefore, *vc[j]* will always be a connected graph at any level. ■

Theorem 1 is useful since, as a consequence, it proof that there is a valid routing algorithm on DCG.

The group of virtual clusters in G_d is called a *Virtual Cluster Set (VCS).* Note that $vc \in VCS$ is a strongly connected component in *DCG*. Virtual clusters are represented as a graph called a *Virtual Cluster Graph (VCG)* that defined in Definition 9.

Definition 9 The virtual cluster graph (VCG)

The *virtual cluster graph (VCG)* $G_v = (V_v, E_v)$ where V_v is a set of virtual clusters. An edge $(vc_1,vc_2) \in E_3$ if and only if $\exists u \in V_d$, such that $u \in vc_1$ and $u \in vc_2$.

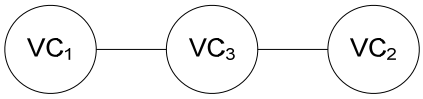

Figure 14 A virtual cluster graph of Figure 10

The longest path in a VCG equals the longest path length in G_d. After a VCG is created, instead of using an inefficient hierarchical routing discovery, a virtual cluster model allows a much better routing as described in the following section.

Proposed Routing Discovery Algorithm

Instead of creating routes from each node in DCG, routes are created from VCG in order to reduce a routing discovery time from the number of node to the number of VC. At virtual cluster level, a route from vc_1 to vc_2 is a simple path $<c_0, c_1, \ldots, c_k>$ such that $vc_1 = c_0$ and $vc_2 = c_k$ where $c_1, c_2, \ldots, c_k \in V_v$. The vertex c_1 in the path is called a *next hop VC* of routing from vc_1 to vc_2. The source vertex only needs to know the next hop to a given destination. The next hop VC of each pair of source and destination are maintained in a routing table. Definition 10 defines the routing table formally. (The term *routing matrix* is used interchangeably).

Definition 10 The routing table

The *routing table (T)* is an $n \times n$ matrix where n is the number of vertices in V_v such that $t_{uv} = c_1$.

Floyd-Warshall's algorithm (Cormen *et al.*, 2001) is used to find all paired shortest paths and a routing table. Floyd-Warshall's algorithm needs two data structures: a shortest distance matrix *D,* and a routing matrix *T*. The shortest path matrix *D* is an $n \times n$ matrix where n is the number of vertices in V_v and $d_{uv}^{(k)}$ is the weight of the shortest path from u to v at the k^{th} iteration of the algorithm. Initially, $d_{uv}^{(0)} = 1$ if and only if an edge $(u,v) \in E_v$; otherwise $d_{uv}^{(0)} = \infty$. The routing matrix is

initialized as $t_{uv}^{(0)} = v$ if $(u, v) \in E_3$, otherwise $t_{uv}^{(0)}$ = NIL. The routing table is created together with the shortest distance matrix. The routing discovery algorithm is shown in Algorithm 2. After the routing table is created, a routing function r can be defined as $r(u, v) = t_{uv}$.

Algorithm 2 A routing discovery

```
n ← number of vertex in G2
D(0) ← W
for k ← 1 to n
     do for i ← 1 to n
          do for j ← 1 to n
               do
                    if d_ij^(k-1) ≤ d_ik^(k-1) + d_kj^(k-1) then
                         d_ij^(k) ← d_ij^(k-1)
                         t_ij^(k) ← t_ij^(k-1)
                    else
                         d_ij^(k) ← d_ik^(k-1) + d_kj^(k-1)
                         t_ij^(k) ← t_ij^(k-1)
                    end if
return
```

An MPI application consists of a group of processes running cooperatively. In our implementation, an MPI process is identified by a *MPI process ID (MPID)* that is equal to a rank in MPI_COMM_WORLD. So, the routing used at runtime must be indexed by MPID. The mapping between an MPID and a node ID is known prior to the start up of MPI tasks. The mapping rule is that at least one process must be mapped onto a gateway node of each cluster to provide the necessary routing capability. Formally, given a DCG $G_d = (V_d, E_d)$ that represents a Grid, and VCG $G_v = (V_v, E_v)$, process mapping can be defined as follows:

Definition 11 **The process mapping**

The *process mapping* is a function $m: P \rightarrow V_d$ where P is a set of processes and V_d is a set of vertices in G_d.

Definition 12 **The reverse of the process mapping**

The *reverse of the process mapping* is a relation $\mu: V_d \rightarrow P$ where V_2 is a set of vertices in G_d and P is a set of processes.

Definition 13 **The node mapping**

The *node mapping* is a function $N: V_d \rightarrow P(V_v)$ where V_d is a set of nodes and $P(V_v)$ is a power set of VC in G_v.

The message routing from process p_1 to p_2 is implemented as followed: First, nodes that p_1 and p_2 belong to are determined. Let $m(p_1) = u$ and $m(p_2) = v$. Then, VCs that u and v belong to are calculated from the function N. Let $N(u) = s_1$ and $N(v) = s_2$. If $s_1 \cap s_2 \neq \varnothing$, u and v are in the same VC. So, p_1 and p_2 are also in the same VC. This means that, by definition of VC, p_1 can initiate a connection to p_2 directly.

The message routing finishes. If $s_1 \cap s_2 = \varnothing$, p_1 and p_2 are not in the same VC; the message must be routed through some other nodes. A next hop VC is selected from the routing table. Let *nvc* be a next hop VC. An *nvc* is selected given that $\forall c \in s_1$, $nvc \neq c$. After the *nvc* is selected, a next hop node is determined by finding a node u_1 such that $u_1 \in nvc$ and $N(u_1) \cap s_1 \neq \varnothing$, which is, informally, find a node u_1 that is in the same VC as u. After the next hop node is determined, the process that is mapped onto u_1 can be identified. Suppose that $\mu(u_1) = p_3$. At this point, p_1 creates an MPI message and sends to p_3, specifying that the destination is p_2. After p_3 receives message, it uses the same method to find a next hop process. This forwarding routine stops when p_2 receives the message.

An example routing scenario is process 2 which is mapped onto Jupiter2 when it wants to send a message to process 3 which is mapped onto Venus1 using the routing matrix, the process mapping, and the node mapping shown in Table 2, Table 3, and Table 4, respectively. In this example, process 2 is running on Jupiter2 which is mapped to VC_2, and process 3 is running on Venus1 which is mapped to vc_1, and vc_3. Process 1 looks for a next hop VC in the routing table for the target vc_1, and vc_3. It finds that the next hop VC is vc_3. It looks for a node that is in both vc_1 and vc_3. There exists a node Jupiter1 which matches the requirement. Then it finds a process mapped onto Jupiter1 that is process 1. Then, process 2 creates an MPITH message identifying the destination of process 3 in the message header, and sends the message to process 1. When process 1 receives the message, it knows that the destination process 3 is mapped onto Venus1 which is in the same VC as itself. Then process 1 forwards the message to process 3. When process 3 receives the message, it finds that the message destination is itself. It saves the incoming message into its buffer and the message relaying finishes. Figure 15 shows this relay routing.

Table 2 An example of a routing table of three virtual clusters under the proposed virtual cluster model

Source \ Destination	VC1	VC2	VC3
VC1	VC1	VC3	VC3
VC2	VC3	VC2	VC3
VC3	VC1	VC2	VC3

Table 3 An example of a process mapping

Process	Node
1	Jupiter1
2	Jupiter2
3	Venus1
4	Venus2
5	Mercury1
6	Mercury2

Table 4 An example of a node mapping

Node	Virtual cluster set
Jupiter1	$\{VC_2, VC_3\}$
Jupiter2	$\{VC_2\}$
Venus1	$\{VC_1, VC_3\}$
Venus2	$\{VC_1\}$
Mercury1	$\{VC_3\}$
Mercury2	$\{VC_3\}$

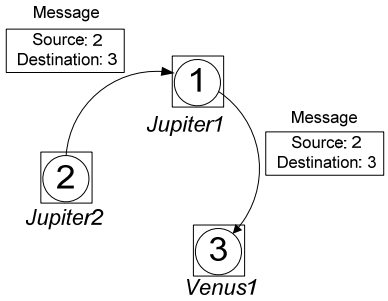

Figure 15 An example of routing scenario

Multicast Operations Model

Multicast operations are the communications among a set of processes. The completion of a multicast operation is specified by a *total transmission time* that is a time between the first process sending data and the last process receiving data. Given a multicast operation, there are a number of data transmissions between sending and receiving processes. Each data transmission in a multicast operation is called a *multicast task* that is defined formally in Definition 14.

Definition 14 The multicast task

The *multicast task (t)* is an order pair $t = (src,des)$ where *src* and *des* are a source and destination process, respectively.

Multicast tasks express an organization of senders and receivers in its operation. On the other hand, they represent a topology within a multicast operation. That topology is called a *multicast topology* defined in Definition 15.

Definition 15 The multicast topology

The multicast topology is a directed graph $G_T = (V_T,E_T)$ where V_T is a set of nodes and E is a set of edges representation of data transmission between nodes.

For a broadcast operation, a multicast topology is a tree that the originator of the data is a root node and all nodes involved in the operation are members of this tree. Because of the fact that bandwidth in a Grid system varies widely, different orders of transmissions within a multicast topology yield a substantially different total

transmission time. So, the order of transmission is another important part and must be considered. Definition 16 defines a multicast order formally. From the definition, a task t_i must begin after a task t_j has finished if $\Omega(t_i) > \Omega(t_j)$.

Definition 16 **The multicast order**

The *multicast order* is a function Ω: $S \rightarrow Z$ such that for $t_i = (s_1, r_1)$, $t_j = (s_1, r_2)$, $t_i, t_j \in S$; $\Omega(t_i) < \Omega(t_j)$ if and only if s_1 sent message to r_1 before it sends to $r2$.

For a given multicast operation, there is a number of solutions to complete the operation. Each solution contains a different multicast topology and order. The combination of a multicast topology and order is called a *multicast schedule*. Definition 17 defines a multicast schedule formally. Figure 16 and Table 5 show examples of a multicast topology and order of a broadcast operation, respectively. There are many feasible multicast schedules which can result in a different total transmission time of the multicast operation. So, the problem addressed here is how to find the best performance multicast schedule after a location of processes are known.

Definition 17 **A multicast schedule**

A *multicast schedule (S)* is an order pair (G_T, Ω) where G_T is a multicast topology and Ω a multicast order.

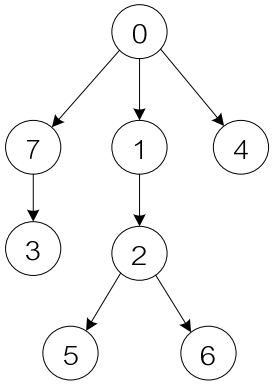

Figure 16 **A multicast topology**

Table 5 **A multicast order**

Order	Multicast task
1	(0,7)
2	(0,1)
3	(0,4)
4	(7,3)
5	(1,2)
6	(2,5)
7	(2,6)

Genetic Algorithms-based Dynamic Tree Algorithm

A proposed Genetic Algorithms-based Dynamic Tree (GADT) is a multicast schedule generating algorithm based on a genetic algorithm. GADT uses a total transmission time as a fitness value. To solve this problem using genetic algorithm, the problem is encoded as follows:

A DNA string is a concatenation of two arrays. The index of these arrays starting from one. The first array is called parent array that represents a multicast topology. The value at index *i* of the parent array, denoted by *p[i]*, is the value of parent node of node number *i*. There are only *n - 1* entries required for *n* nodes because node *0* does not have any parent. The other one is a priority array that represents a multicast order. The priority values are used by parent nodes to choose which nodes to send data first. The priority value begins from 0 to a predefined value, which is 1,000,000 in the experiment. A lower value means a higher priority. If two nodes have the same parent and priority, the lower node number will be chosen first. Figure 17 and Figure 18 show an example of multicast schedule for eight nodes and a GADT encoded string, respectively.

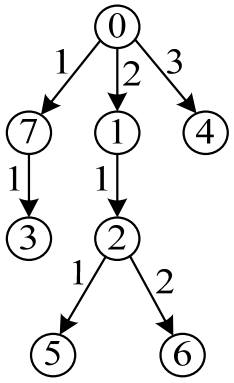

Figure 17 An example of multicast schedule

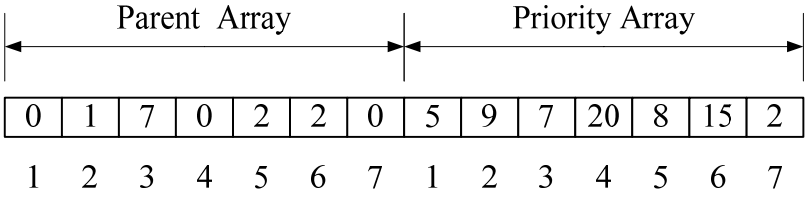

Figure 18 A GADT DNA encoding from Figure 17

The DNA string is a view of a child node. When a multicast tree is formed, the process starts at node 0 then continually looks for the next child. The DNA string is converted to a *child matrix* which is $n \times (n-1)$. The i^{th} row are the children of node *i*. The order in each row is an order of sent data. Figure 19 shows a child table generated from DNA in Figure 17.

7	1	4	0	0	0	0	0
2	0	0	0	0	0	0	0
5	6	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0

Figure 19 **A child table generated from Figure 18**

Because of the fact that GA performs a random search, crossing over and mutation, some processes may not have any parent or a multicast topology may consists of more than one connected component. In this case, some nodes cannot receive data from the root. The topology must be checked before it is evaluated for fitness value by a simulator. Figure 20 and Figure 21 show an example of DNA for this problem, and a multicast schedule generated from the DNA, respectively. Algorithm 3 shows an algorithm for detecting unconnected nodes and looping links.

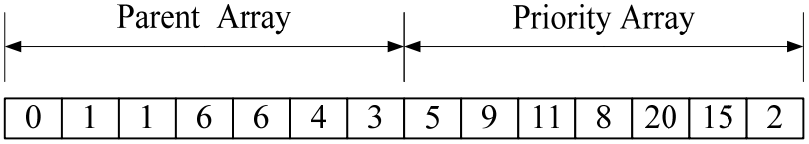

Figure 20 **An example of invalid DNA**

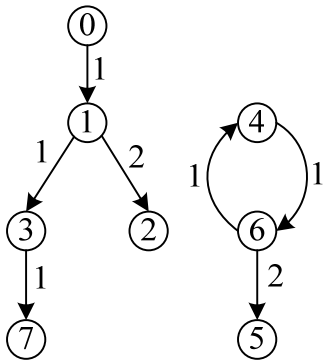

Figure 21 **A multicast schedule generated from Figure 20**

Algorithm 3 A node detection

```
     input: childtable, nodeno, depth, connectingtable
     output: connecingtable, rc
begin
     OK ← 0
```

```
     LOOP ← 1
     rc ← OK

     if depth > nproc then
          rc ←LOOP
          exit
     end if

     for each i in child of node number nno do
          if i = 0 then
               continue
          else if connectingtable[i] = true then
               rc += LOOP
          else
               connectingtable[i] ← true;
               rc += NodeDetection(childtable, i, depth + \
                    1, connectingable)
          end if
     end for
     return rc;
end
```

The node detection is a recursive algorithm. It begins with a given child table, a node number 0, a connecting table. In the connecting table, only node 0 is marked as *true*, which means connected, and other node is marked as *false* and a current depth. The output of this algorithm is the connect table and the number of loops. In this algorithm, *rc* is a return code which is initialized to *OK*. To prevent infinite loop, *depth* variable is checked. If *depth* is greater than the number of node, this means that the algorithm sticks in a loop. Then, for each child of current node, it checks whether this child has been already visited. If not, it marks this child as connected node. After that, it recursives to this node by incrementing the depth parameter by one. The number of unconnecting nodes can be computed from connecting table.

After the connecting table is created, a precomputed fitness value of a DNA string is computed. Algorithm 4 shows this precomputed algorithm. This algorithm takes the child matrix and the number of nodes and returns a precomputed fitness value. At the beginning, it initializes the connecting table, call *NodeDetection* to detect unconnected nodes and loops. If an unconnected node or a loop is detected, the predefined fitness value is set to 1000 times the maximum transmission time computed from $\frac{data\ size}{min\ bandwidth} \times (number\ of\ node - 1)$. This is a bias such that a bad multicast pattern will have a much higher chance of being dropped. Then *precompute_fitness* is added by loop and unconnected penalty.

Algorithm 4 A DNA precomputed

```
     input: childtable, num_node
     output: precompute_fitness
begin
     initialize connectable
     precomptue_fitness ← 0
     loop ← NodeDetection(childtable,0,0,connectedtable)
```

```
     nc ← number of unconnected node
     if loop ≠ 0 or nc ≠ 0 then
          precompute_fitness += max_use_time × 1000
     end if
     precompute_fitness += loop × LOOP_PENALTY
     precompute_fitness += nc × NC_PENALTY
end
```

With Algorithm 3 and Algorithm 4, the evaluating function for PGAPack is shown in Algorithm 5. At the beginning, a childmatrix is created from DNA strings. Next, a *Precomputed* function is called to get a precomputed fitness value. If the precomputed fitness value is not zero, this value is returned because the topology has loops or unconnected nodes. Otherwise, the multicast schedule is created and passed to TreeSim in order to compute a total transmission time. This process continues until the near optimal answer is obtained or a predefined number of iteration is reached. Figure 22 shows an overview of the GADT algorithm.

Algorithm 5 DNAEvaluate

```
     input: dnastring, number_of_node
     output: fitness_value
begin
     create childtable
     fitness_value ← precompute(childtable, number_of_node)
     if fitness_value ≠ 0 then
          exit
     else
          create schedule for tree evaluator
          fitness_value ← tree_evaluate(schedule)
          exit
end
```

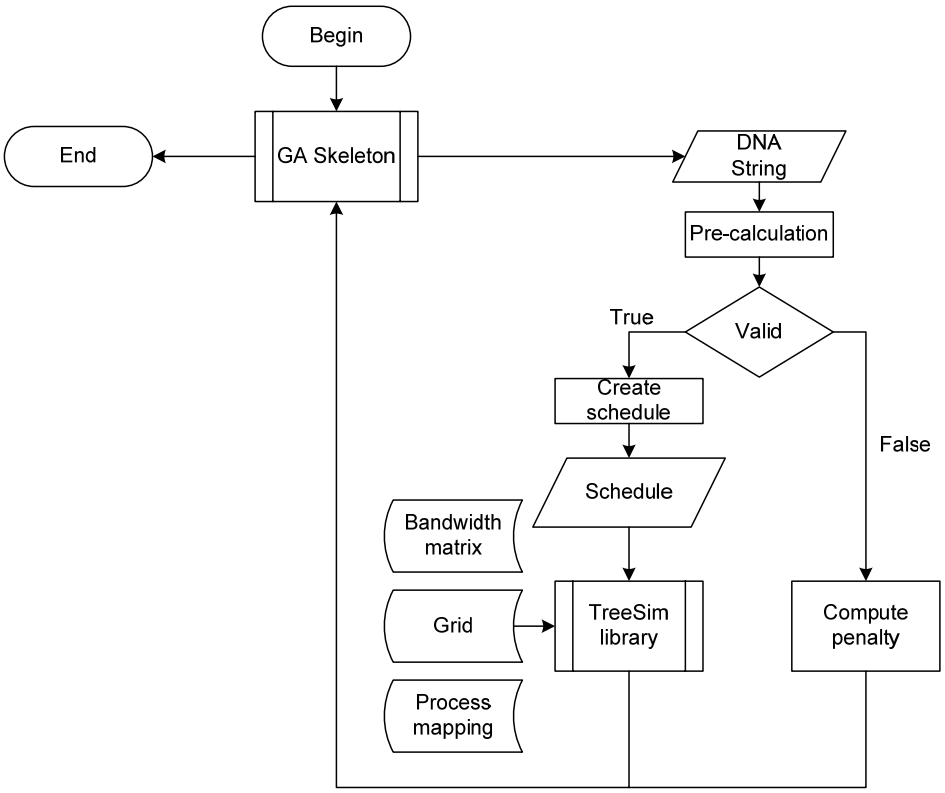

Figure 22 A GADT algorithm

Longest Parallel Branch First Algorithm

The GADT algorithm requires a long computation time. It is not suitable for using in MPI runtime library. This section describes a new heuristic algorithm named *Longest Parallel Branch First (LPBF)* algorithm. It is an online algorithm that can be built into an MPI runtime library. The concept is based on the efficient topology selection used in ECEF coupled with the intelligent message scheduling. This algorithm works as follows:

1. The topology of the inter-cluster is created from a Grid graph by representing each cluster as a node. Then, the ECEF algorithm is used to produce a intercluster multicast topology. The intention of this step is to minimize intercluster communication by disallowing the direct communication among nodes in multiple clusters.

2. Within each cluster, the binomial algorithm is used to generate a multicast schedule in order to obtain a very close to optimal communication performance for each system.

3. The multicast order is then created by computing the total transmission time of each branch from the current node and selecting the longest branch as the first target. This step is intended to exploit the fact that the size of each target cluster can

be very different. Hence, some concurrency in message transmission can be exploited. This step is repeated until all leave nodes of the multicast topology are reached.

The concept used here is based on the fact that the longest branch usually belongs to the part of a Grid with more nodes. Thus, scheduling a transmission on that branch first can help maximize the overlapping communication. Moreover, this simple concept can well capture the fact that the size of the clusters on a large grid can vary significantly. Figure 23 through Figure 25 show the operation of the LPBF algorithm.

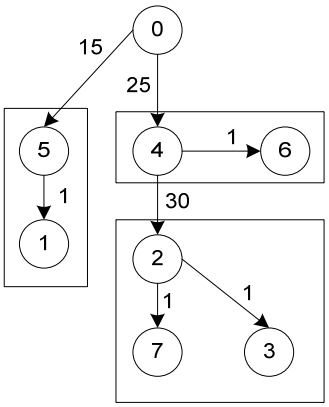

Figure 23 A multicast tree obtained from the first two steps of LPBF algorithm

Figure 23 shows multicast topology obtained from the first two steps. The number associated with link is a transmission time of that link. The topology is imported to the LPBF algorithm. LPBF computes branch times, shown in braces. For example, node 0 has two children, 5 and 4. The left child has branch time one time unit because node 5 uses only one time unit to send to node 1. Branch time for node 4 is 32 time units because it sends data to node 2 using 30 time units. Then, node 2 completes its children using two time units. The time that node 4 sends to 6 is not added to value 32 because this time overlaps with the time that node 2 completes its children. The output of this step is shown in Figure 24.

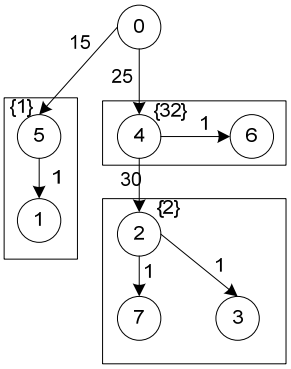

Figure 24 Branch times

An ordered pair *(a,b)* denotes a start and finish time of a link's communication. This figure shows that node 0 sends data to node 4 before node 5 because node 4 has a branch time more than node 5's.

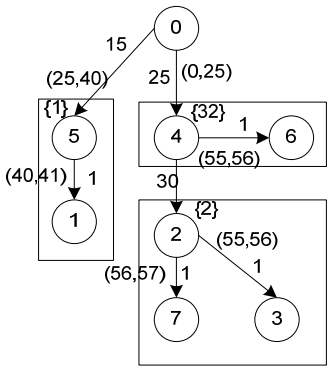

Figure 25 An LPBF schedule

Algorithm 6 shows a main LPBF algorithm. It takes a bandwidth matrix, and a process mapping and returns a multicast schedule. The algorithm creates a multicast schedule from ECEF algorithm at a cluster level. Then, multicast topology is extracted from the returned schedule. After that, it calls LPBF_schedule function to create an LPBF schedule based on the ECEF topology starting at the root node, which is node 0.

Algorithm 6 LPBFMain

```
     input: bwmtx, process_mapping
     output: lpbf_sched
begin
     tschedule ← ecef_two_level(bwmtx, process_mapping)
     topology ← get_topology(tschedule)
     lpbf_sched ← lpbf_core(topology, 0, bwmtx, lpbf_sched)
     return lpbf_sched
end
```

Algorithm 7 shows an LPBF scheduling algorithm. It takes a multicast topology, a current node, a bandwidth matrix, and a current schedule, which is to be filled. LPBF_schedule recursively gets a branch time of each branch by calling itself on each child. The branch time is saved along with each child. Then, children are ascendingly sorted by their branch time. A multicast task is created from a current node to each of its children and a transmission time is computed. The current time is updated by accumulating the transmission time of each branch. The multicast task is added to the multicast schedule, which is `currschedule` in the code. Finally, a current time is returned as a branch time of this branch root at this node.

Algorithm 7 LPBFSchedule

```
     input: topology, currnode, bwmtx, currschedule
     output: currschedule, branch_time
begin
```

```
     my_children ← get_children(currnode, topology)
     for all child ∈ my_children
          child.branch_time ← child.lpbf_core(topology, child, bwmtx,
currschedule)
     end for

     sort(my_children)
     currtime ← 0
     for all child ∈ my_children
          t ← get_transmission_time(currnode, child) +
child.branch_time
          currtime ← currtime + t
          add_schedule(currnode, child, currschedule)
     end for
     return currtime;
end
```

Tree Simulator

TreeSim (Vorakosit and Uthayopas, 2005b) is a library module used to evaluate proposed multicast algorithms. It computes a total transmission time from given parameters and returns a total transmission time to the caller. Figure 26 shows an overall TreeSim architecture.

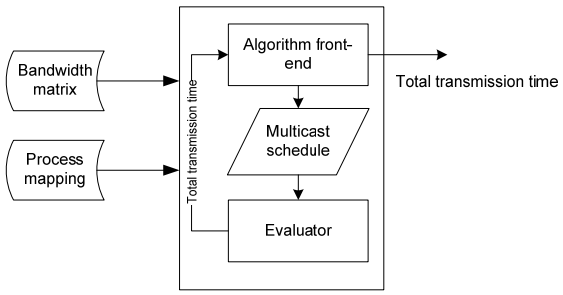

Figure 26 A TreeSim architecture

From Figure 26, the simulator tool consists of two main modules: an *algorithm front-end* and an *evaluator*. The algorithm front-end generates a multicast schedule from a bandwidth matrix and Grid information. The algorithm front-end produces and sends the multicast schedule to an evaluator and receives a total transmission time back from the evaluator. After that, it may exit immediately or produce the next input to evaluator depends on specific algorithm it is.

An evaluator function calculates the total transmission time for one instance of the multicast schedule. It uses a bandwidth matrix, a process mapping table, and a multicast schedule as input. The overall evaluator's algorithm is shown in Figure 27.

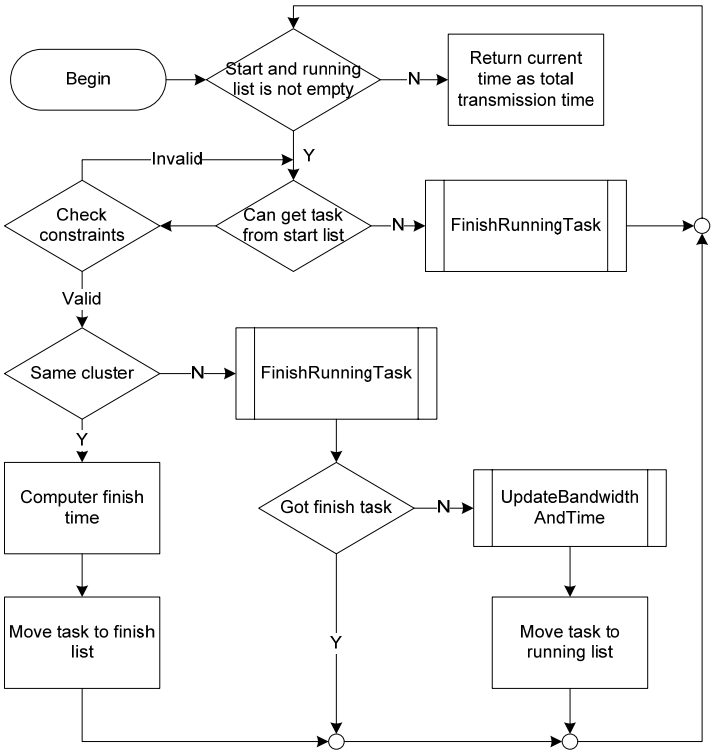

Figure 27 An evaluator's algorithm

The evaluator has three lists: start, running and finish. Each list keeps tasks in started, running and finished state, respectively. The runtime information of processes is maintained in a process list. The process list is derived from a process set. Each entry contains: ID, node ID, cluster ID, free time, busy flag and data flag. Free time is the time that this node can start executing a next task. A data flag is set to *true* if the process already has data; otherwise it is set to *false*.

The overall algorithm is as followed: It gets a task from the start list. For each selected tasks, it checks whether source process have already acquired data and whether source and destination processes are busy. If there is any condition that does not satisfy these constrains, it skips the selected task by moving the task back to the start list and continues to process the list. If the source and the destination processes are in the same cluster, this kind of task is ready to finish. The transmission time is computed from the bandwidth in that cluster. The data flag for the destination process is set to *true*. The finish time of the task is calculated from the transmission time plus sender's free time. The free times of both processes are updated to finish time. Finally, the finished task is moved to the finish list.

If source and destination processes are not in the same cluster, the algorithm is more complex. Based on its start time, it searches whether there is any finish task before the start time. If so, the earliest finished task is moved to the finish list. This function is shown as *FinishRunningTask* in Figure 27. After that, this task is moved back to the start list and the evaluator continues to get the next task from start list. This is a result of the process associated with this finish task must start its next task

immediately. If there is no finish task, it updates the bandwidth and estimated transmission time of tasks that use the same link as this current task. This is done by *UpdateBandwidthAndTime*. Then, this task is moved to the running list.

When an algorithm cannot get any tasks from the start list, it gets the earliest finished task from running list by calling *FinishRunningTask* function to move the finished task to finish list. After that it goes back to select tasks from the start list again. The algorithm ends when all tasks are moved to the finish list.

MPITH Design and Implementation

1. Architecture

The MPITH architecture is as illustrated in Figure 28. MPITH is composed of four layers: device, device handler, engine, and API layers; each of these is described below:

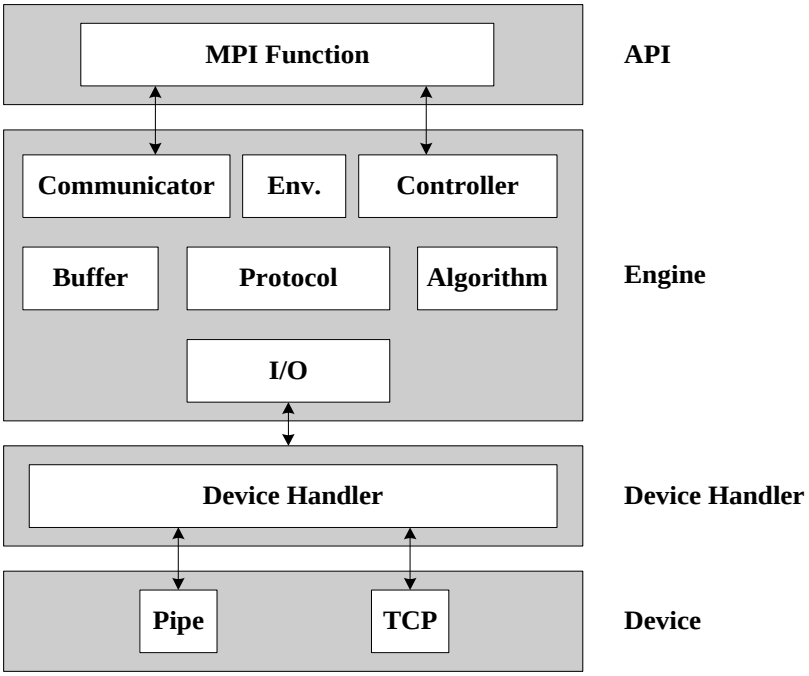

Figure 28 The MPITH architecture

1.1 The device layer

MPITH is designed to support multiple underlying network protocols through the use of a separate device layer that is responsible for interfacing with the network protocols, such as TCP and UDP. A device is categorized as either client mode or server mode; a client mode device is responsible for connection initialization and data transmission between MPI processes. A server mode device waits for client

connections. When a server receives a new connection, it creates a new client mode device to handle the connection. The device layer has the following requirements:

- Reliable transmission
- Supports the *select()* system call
- Operates in synchronous mode but support both blocking and non-blocking operation

1.2 The device handler layer

The device handler layer is responsible for runtime device management. It maintains a server device and client devices. The client devices are indexed by an MPI process ID (MPID). The engine layer module sends/receives data through this layer except at the beginning of the program execution. This layer manages devices' status and guarantees that the engine layer can send and receive data at any time regardless of the status of devices. This strategy allows the device to stay open, or connected, most of the time in order to reduce the latency for connection initialization.

1.3 The engine layer

The engine layer is the core of the MPI library. It services requests from the API layer, handles incoming data from the device handler layer, and executes communication algorithms such as MPI_Bcast and MPI_Reduce. The engine layer is separated from the API layer to permit easier integration of new features into the API layer, such as message logging and profiling instructions. The engine layer contains seven modules:

- I/O module. The I/O module is the lowest module in the engine layer. The I/O module is responsible for reading and writing data from/to the device layer. For incoming data, it calls *select()* to select a readable device and reads the data. If the final destination of an incoming message is itself, the message is stored in the buffer module; otherwise, it will ask the environment module for a next hop of the destination and puts the message in its internal buffer. Outgoing data consists of messages generated by non-blocking MPI functions, such as MPI_Isend, plus forwarded messages as described above. For output, the I/O module selects writable device using *select()*, and sends the data through that device.

- Protocol module. The protocol module is responsible for packing an MPI request into an MPI segment to be sent through the I/O module. The protocol module connects to three modules: the I/O module to perform data transmission; a buffer module to save incoming data or load outgoing data from asynchronous MPI functions; and the communicator module to service request MPI function.

- Buffer module. The buffer module buffers incoming and outgoing data until it can be passed to other modules. Incoming data is read from the device handler by the I/O module and placed in a buffer until the user-level application requests data. Outgoing data is buffered if it is generated by non-blocking MPI functions. A buffered segment is sent when there is no other pending outgoing segment. The buffer module provides both blocking and non-blocking data retrieval and search functions.

- Algorithm module. The algorithm module is responsible for creating a multicast communication schedule for multicast operations. The communication schedule is a series of parents, and children in multicast topology. The communicator module uses the schedule to prepare an outgoing message buffer and sends it to the protocol module.

- Communicator module. The communicator module is responsible for servicing MPI functions that involve user data transmission. Point-to-point and collective MPI functions call this module. For outgoing functions such as MPI_Send or MPI_Bcast, if the request is a collective communication function, it consults the algorithm modules to create a multicast schedule. In some case, data is rearranged in this module for ease of the sending operation. For example, MPI_Scatter sends data to each destination based on MPI rank but the schedule received from the algorithm module does not order as user data. After that, outgoing data is sent to protocol module to perform a formatting and then is sent to a destination. For an incoming function, such as MPI_Recv or MPI_Reduce, it gets data for an MPI segment in the buffer module. Internally, the communicator module maintains a list of *MPI communicators*. The communicator is a mapping between rank and MPID as well as its context.

- Control module. The control module is responsible for handling MPI functions about informative and administrative task.

- Environment module. The environment module is responsible for managing an environment configuration and a process mapping. The environment configuration describes a host list and a cluster connectivity. It is used to create a routing table at node level. The process mapping maintains a mapping between MPI process ID and node ID that the process is running on. The process mapping is used to create a routing table indexed by an MPI process ID. This routing table is used when the MPITH process wants to send or forward message.

1.4 API layer

The API layer interfaces with the application programmer. The syntax of API conforms to subset of MPI standard. MPITH provides both C++ and C interfaces. The 67 functions supported by MPITH are as follows:

1. Point-to-point communication. There are two modes:

a. Blocking mode. In blocking mode, for sending side, message must be saved in the receiver's buffer before MPI function can return, and for receiving site, data must be in user's buffer before MPI function returns. There are six functions in this mode: MPI_Send, MPI_Recv, MPI_Ssend, MPI_Sendrecv, MPI_Sendrecv_replace, and MPI_Probe

b. Non-blocking mode. In this mode, the MPI function returns immediately, users have to manually check whether the operation is complete. There are four functions in this mode: MPI_Isend, MPI_Issend, MPI_Irecv and MPI_Iprobe

2. Collective communication – six functions: MPI_Bcast, MPI_Reduce, MPI_Scatter, MPI_Gather, MPI_Barrier, and MPI_Allgather

3. Communicator management – ten functions: MPI_Comm_dup, MPI_Comm_split, MPI_Comm_free, MPI_Get_rank, MPI_Get_size, MPI_Group_incl, MPI_Group_excl, MPI_Group_size, MPI_Group_rank, and MPI_Group_compare

4. Environment management – eight functions: MPI_Init, MPI_Finalize, MPI_Initialized, MPI_Finalized, MPI_Abort, MPI_Get_processor_name, MPI_Wtime, and MPI_Wtick

5. Data operation - two functions: MPI_Op_create, and MPI_Op_free

6. Request Management – nine functions: MPI_Wait, MPI_Waitall, MPI_Waitsome, MPI_Waitany, MPI_Test, MPI_Testall, MPI_Testsome, MPI_Testany and MPI_Free

7. Data type management – 15 functions: MPI_Type_commit, MPI_Type_free, MPI_Type_size, MPI_Pack, MPI_Pack_size, MPI_Type_commit, MPI_Type_contiguous, MPI_Type_extent, MPI_Type_free, MPI_Type_hindexed, MPI_Type_hindexed, MPI_Type_hvector, MPI_Type_indexed, MPI_Type_lb, MPI_Type_size, MPI_Type_struct, MPI_Type_ub, MPI_Type_vector, and MPI_Unpack

8. Miscellaneous functions – three functions: MPI_Address, MPI_Get_count, and MPI_Get_elements

2. Process creation

There are three process types: a primary master, a secondary master, and a slave mode. The *master*-type process is responsible for spawning of processes within its physical cluster. The primary master process differs from the secondary in that the former is a source of configuration files. A process that is not a master-type process is a *slave* process. All process types have the same execution codes. The process type is specified in configuration files. There are three configuration files: a grid configuration file, and a process mapping file, and a bandwidth file.

On startup, master-type processes are started in each cluster using *mpirun* or *globusrun* depending on the application type. The *mpirun* command is for running MPI applications within a single cluster. In this case, the user tells *mpirun* the number of processes and a host list file to run application; these arguments are converted to the configuration files as described above. In case of a Grid, user first uses *mpithrsltemplate* to create a template of RSL script for this application. The user may edit this file to match his needs, for example, the command-line argument, an environment, etc. Then, the user executes the command with the RSL script using globusrun. The primary master process has two additional configuration files as arguments.

After these master processes have started, the MPI_Init function is executed. In MPI_Init, the master processes synchronize using DUROC and learn the IP address/port of all other master processes. Then, the primary master process broadcasts the configuration files to the other masters. After the secondary master processes receive the configuration files, they spawn child processes in their cluster according to the requirements in the process mapping file. Child processes then execute the MPI_Init function to register themselves with their master. Then, all processes return from MPI_Init and execute user code. Figure 29 shows a process startup sequence.

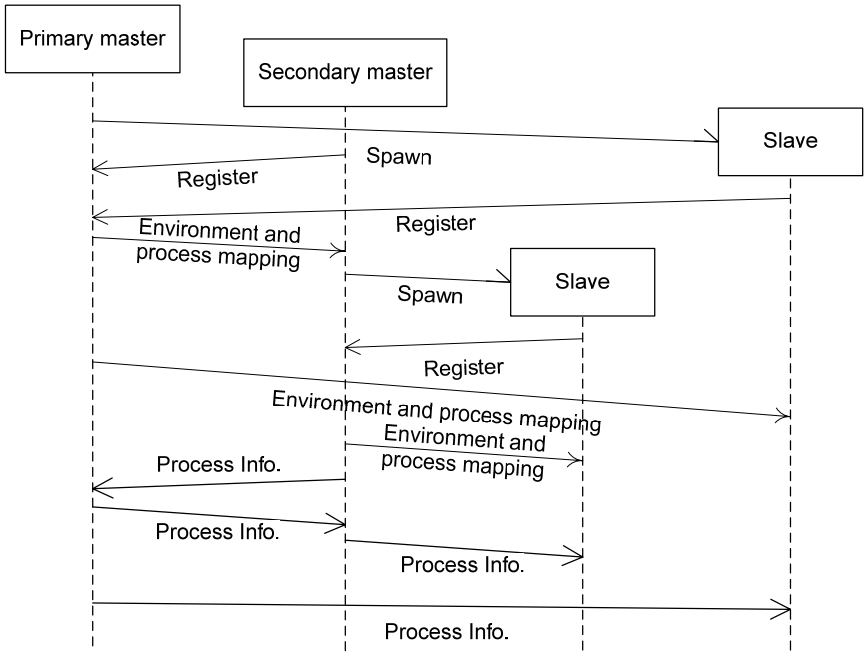

Figure 29 A process startup sequence

3. The MPITH Message format

Figure 30 shows a format of MPITH message. It is composed of six fields. The first field is a command or message type that is a 16-bit field. An MPITH command is divided into three categories: data, control and self control. The data

message is sent from an MPI function. The control message is used to send the runtime information to the other process. The self control message is a control message that targets itself. The second field is a context used to specify the communicator. The next field specifies the size of a payload in bytes. The next two fields are destination and source, respectively. The last field is a payload of MPI message that can be omitted. Each MPITH command has its own format of payload. For example, a payload of MPI_Send command contains a data type, and a tag field while a payload of MPI_Bcast command contains a data type, and a root field.

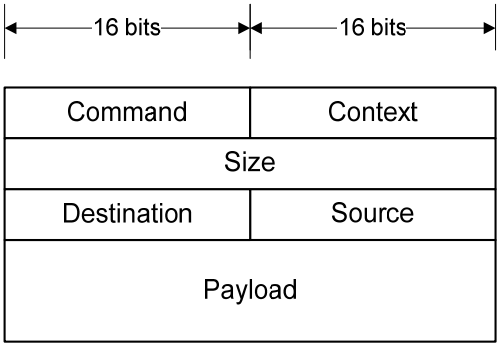

Figure 30 An MPITH message format

4. The Buffer Management Module

The buffer management module is responsible for maintaining both incoming and outgoing buffer. The incoming buffer receives data from the I/O thread that reads from the device handler module. All incoming messages that the destination is itself are saved in incoming buffer. This improves overall performance of an MPI application because a sender need not to wait for a receiver to issue a receive command. MPI standard has a tag feature that allows a programmer to select a message using a message tag. The incoming buffer is designed to support tag in an MPI message selection. An architecture of incoming buffer module is shown in Figure 31

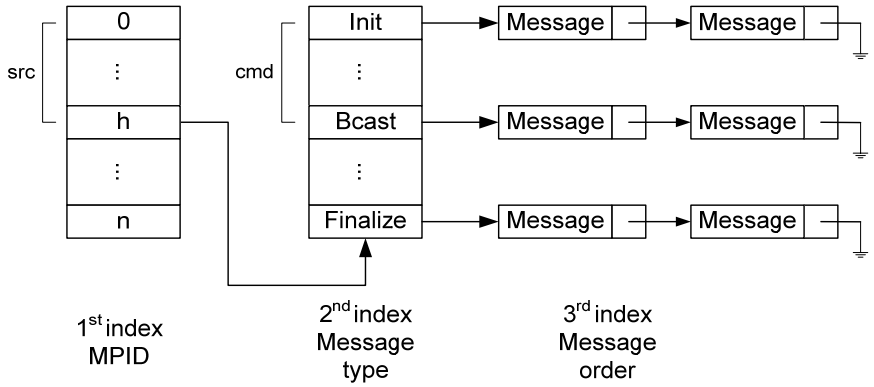

Figure 31 The incoming buffer

The first level is indexed by MPID. The protocol module can query messages by node or MPI_ANY_SOURCE. If protocol selects MPI_ANY_SOURCE, the buffer manager will query every node. The first level index points to the second array that is indexed by command. The entry in the message array points to a message queue that is ordered by incoming order. If a tag is used, a message is selected using tag; otherwise, the first message in the queue is selected. The incoming buffer module returns a pointer to message to the protocol module. After the protocol module finishes using the message, which might have been copies to user buffer or processes other MPI operation, it removes message from the incoming module.

An outgoing buffer is used when the user issues a non-blocking MPI function, for example, MPI_Isend. Figure 32 shows architecture of the outgoing buffer. The outgoing buffer is a two-level data structure. The first level is an array indexed by a next hop of the message. A entry in this array points to a message queue which is ordered by outgoing order. The I/O module selects the first message in a queue and sends to the next hop node.

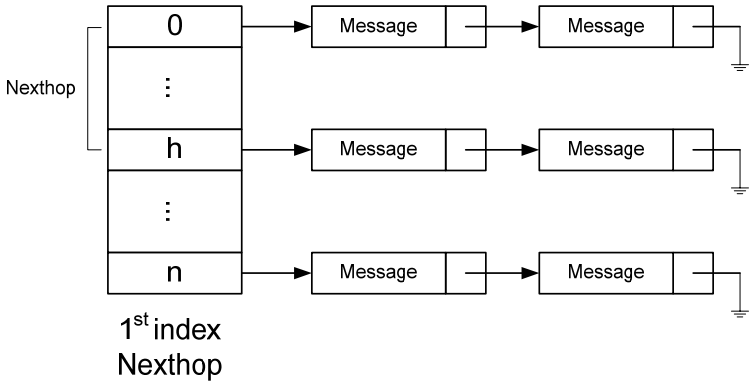

Figure 32 The outgoing buffer

5. Low-Level Message Handling

The I/O module is responsible for reading incoming messages from the device handler module and writing outgoing messages from the outgoing buffer. If a destination of an incoming message is not itself, it must forward the message to the next hop using its routing table. Figure 33 shows architecture of I/O module.

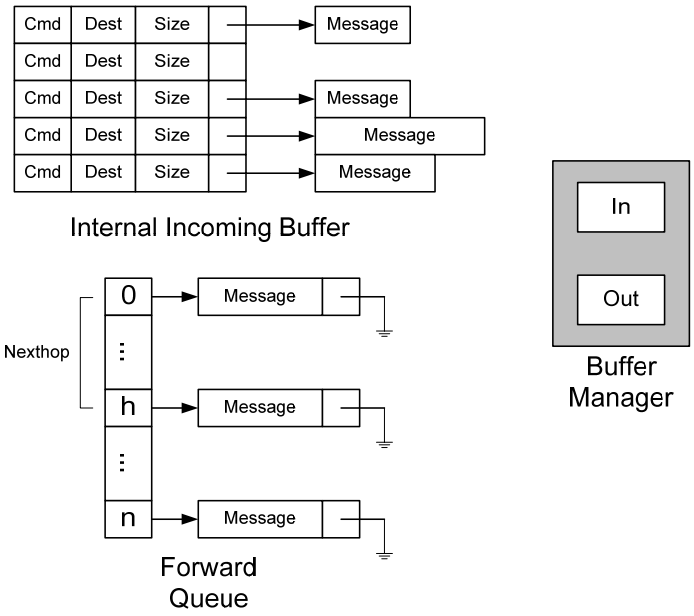

Figure 33 The I/O module architecture

First, it selects a list of ready devices from the device handler module. The selected devices are divided into two categories: ready to receive and ready to send. For the ready to receive device, if this is the first time of this message, it reads 12 bytes of data, which is an MPITH message header and saves the data into its internal buffer. Then it allocates buffer for saving a payload. The destination of the message is checked whether the destination is itself. If not, it pushes this device's ID, which is MPID of the peer process, to the forwarding queue. Then the payload is read and saved to the allocated buffer. The I/O thread reads the payload until it cannot read the data. This means that the message is transmitting over the network. Then, it continues the next device from the ready to receive list. After the message is fully read and the destination is itself, the message is saved in the incoming buffer in the buffer manager module, and process the control message. If the message is not sent to itself, the device used to read the message is marked as *pending*.

After the ready to receive devices have finished processing, it continues the ready to send device. First, it checks whether there is a pending forwarding message for this device. The message is forwarded even though it is not fully read. If yes, it sends the message to this device. The device is locked as *forward* in order to prevent other modules sending data to the device. After the message is fully forwarded, the receiving device marked as pending is changed back to ready. If there is no forwarded message to this device, it checks for an outgoing message from the outgoing queue in the buffer manager module. If there is an outgoing message, it sends the message to this device. The device is locked as *departure* in order to prevent other modules sending data to the device. After the message is fully sent or forwarded, the device is unlocked.

6. A Blocking and a Non-blocking Point-to-point

The blocking operation can be divided into incoming and outgoing operations. MPI functions that use blocking operations include blocking send/receive functions such as MPI_Send, MPI_Recv and collective communication functions such as MPI_Bcast, MPI_Scatter. In an outgoing operation, API layer passes data and destination to the communicator module. If an MPI function is collective communication function, the algorithm is consulted for a multicast schedule. The communicator module arranges an outgoing message and passes it to the protocol module. A target MPID is queried from an MPI communicator. After that, the communicator passes an outgoing message with the target MPID to the protocol module. The protocol module queries a next hop MPID from the environment module. After that, the protocol module sends the message to the next hop process via the device handler layer.

The blocking incoming operation occurs from MPI functions such as an MPI_Recv family, a collective communication family (MPI_Bcast, and MPI_Reduce), and a blocking message query status function family (MPI_Probe, and MPI_Wait). For incoming messages, the I/O module is responsible for reading incoming data and saves it in the buffer module. When the API layer wants data, it calls the communicator module. The communicator module queries MPID from rank in a correspondent MPI communicator. After that, the communicator module uses that MPID to query from the buffer module. If data is found, it is copied to user's memory; otherwise, an application thread suspends itself. When the I/O thread adds incoming data from network into the buffer module, the application thread is woke up. After that, it checks whether a buffer module contains the data it needs. If not, it suspends itself again. Then it gets data and returns from the buffer module and copies data to user's memory.

The non-block operation occurs from MPI_Isend/MPI_Irecv family. In an outgoing operation, the communicator module queries a target MPID, and passes all the information to the protocol module as in the blocking operation. The protocol module queries a next hop MPID. After that, instead of sending a message to the next hop process, the protocol module puts the outgoing message into the outgoing queue in the buffer module. The message is sent later by the I/O module.

The incoming operation includes a non-blocking receive function, namely, MPI_Irecv family, and a message query status function family: MPI_Iprobe, MPI_Test, MPI_Testsome and MPI_Testall. The operation is the same as the blocking outgoing operation except that it returns immediately without blocking if communicator module does not find data in the buffer module.

7. Data Type

MPITH supports 14 instinct data types and 6 derived data types. MPITH provides data packing and unpacking routine. When data is sent, a map of the sent data type in term of instinct data type is created. On the sending side, the map is used

to create I/O vector for *writev* system call, which is a system call that writes multiple blocks of data in one time. This eliminates the use of a packing buffer. On the receiving side, an incoming data is saved in the incoming buffer. A map is used to unpack data in incoming buffer and copy to user buffer.

8. Multicast Operation

MPITH implements collective communication on the top of primitive point-to-point communication. When user invokes the collective communication function, the communicator module consults the algorithm module for a multicast schedule. After that, the communicator performs send/receive operations in the order of tasks in the multicast schedule.

Development Experience

This section discusses experiences of developing MPITH that is about a choice of programming language and debugging tips.

MPITH is developed using C++ language. It is composed of a number of C++ class. This helps the developer to separate each class and test separately. Some modules are developed using Microsoft Visual Studio because it provides sophisticated debugging features. After that, a module is compiled and tested again in Linux environment. There is a little issue when user compiles his parallel program with MPITH; some variables in the source code are a keyword of C++ language, *new* for example. User has to avoid using these keywords.

The challenging problem involved is how to debug the MPITH library. In a single process program, the user can use GDB to step and walk through their code but MPI application is not in that case. An MPI application is composed of a number of processes running in different computers; each process has two threads, which are an I/O and an application thread. A basic *printf*-style debugging is very handy. The test program first redirects standard out and error to a file using *dup2* system call. Then this log file is analyzed to spot faults in source codes. Standard macro `__FILE__` and `__LINE__` and GNU-specific `__func__` macro provide automatic filename, line number, and function name when using with printf.

Another handy utility is *strace*. Strace traces system calls and signals of specified process ID. Strace is used mainly when a program is waiting for some events for an extended period of time. GNU debugger, GDB, is sometimes useful, especially when a program produces a core dump. GDB can display a stack of program at the time the core occur. Sometimes, illegal memory access destroys the stack, GDB cannot display any information. In this case, the program is run again with *Valgrind*, the suite of tools for debugging and profiling. Valgrind reports an illegal memory access and leaks in the program.

Sometimes, unsuccessful running of a program is not a result of a fault in program; it is a result of hardware failure. For example, may be a network cannot handle huge data transfer. This causes a program to wait for a data in an extended period of time. The network failure can be detected by considering the number of data in send queue (Send-Q) of TCP connection using *netstat*. It is a count of bytes not acknowledge by a remote host. If there is a data in sent queue in one host but there is no data in received queue (Recv-Q) of the corresponding TCP connection in the peer host, the problem may be a result of network failure. In this case, *tcpdump* is used to detect whether there is a retransmitted package between those hosts. If there is, the problem is the result of network failure.

RESULTS AND DISCUSSION

This chapter illustrates the performance of proposed algorithms. The proposed multicast algorithms are tested using a simulator. Then, LPBF and VC model are implemented in MPITH in order to be tested in a real Grid environment. This chapter first describes the simulated results of the proposed multicast algorithm. After that the performance of MPITH is shown.

Multicast Algorithms Performance

The proposed algorithms, GADT and LPBF, are evaluated using TreeSim. They are compared to a binomial tree algorithm, ECEF, and an optimal algorithm, which computes a schedule using brute force algorithm. Algorithms are evaluated on Intel Xeon II 3.2 GHz, 4 GB RAM. In this test, two bandwidth matrices for 32 clusters, one is symmetric and the other is asymmetric, are randomly generated using Linux's *rand()* function and a time at the generating is used as a seed. Grid sizes, which are the number of clusters in a simulated environment, are 2, 8, 16, and 32 clusters. The number of nodes in each clusters of each Grid size is randomly generated. Table 6 shows the number of nodes in each cluster for each Grid size. For each test, ten patterns of nodes are randomly selected from a given Grid size.

Table 6 The number of node for two clusters

Cluster	Number of nodes in Grid				
	2	**4**	**8**	**16**	**32**
0	654	611	377	977	790
1	641	465	465	465	465
2		431	431	431	431
3		300	300	300	300
4			611	611	611
5			619	619	619
6			151	151	151
7			270	270	270
8				377	377
9				531	531
10				778	778
11				488	488
12				906	906
13				439	439
14				440	440
15				945	945
16					977
17					617
18					886
19					325
20					737
21					634
22					653
23					424
24					1004
25					745
26					698
27					376
28					963
29					325
30					957
31					188

PGAPack 1.0 library is used as a skeleton of a genetic algorithm. GA specific values are shown in Table 7.

Table 7 PGA pack parameters of test environment

GA Specific Value	Value
Population size	100
Maximum iteration	50000
Stopping criteria	Maximum iteration, or population does not change within 300 iterations
Crossover type	Two points crossover
Crossover probability	0.85
Mutation probability	0.01

First, multicast schedules of eight nodes are created. The number of nodes is fixed to eight because of the very long running time of the optimal algorithm. Table 8 and Table 9 show the total transmission time of eight nodes of symmetric and asymmetric bandwidth, respectively. Table 10 and Table 11 show the normalized total transmission time of Table 8 and Table 9, respectively.

Table 8 Total transmission time of eight nodes, a symmetric bandwidth

Grid Size	Algorithm				
	Optimal	Binomial	ECEF	GADT	LPBF
2	52533.37	130926.75	116846.74	52533.37	52933.37
4	4345.17	72013.74	7915.03	4493.99	5001.98
8	4220.55	45494.95	5274.24	4503.35	5180.04
16	4798.14	90436.20	5474.76	5786.65	5065.62
32	4699.53	58228.83	5505.54	5396.93	5404.07

Table 9 Total transmission time of eight nodes, an asymmetric bandwidth

Grid Size	Algorithm				
	Optimal	Binomial	ECEF	GADT	LPBF
2	18210.04	48131.08	34420.08	18233.05	18841.03
4	3754.42	53209.19	6390.75	4194.79	4549.79
8	4455.07	44485.91	5678.50	4945.86	5574.09
16	4644.27	95466.24	5894.50	5155.92	4943.15
32	5844.55	37153.05	6358.72	6076.36	6151.97

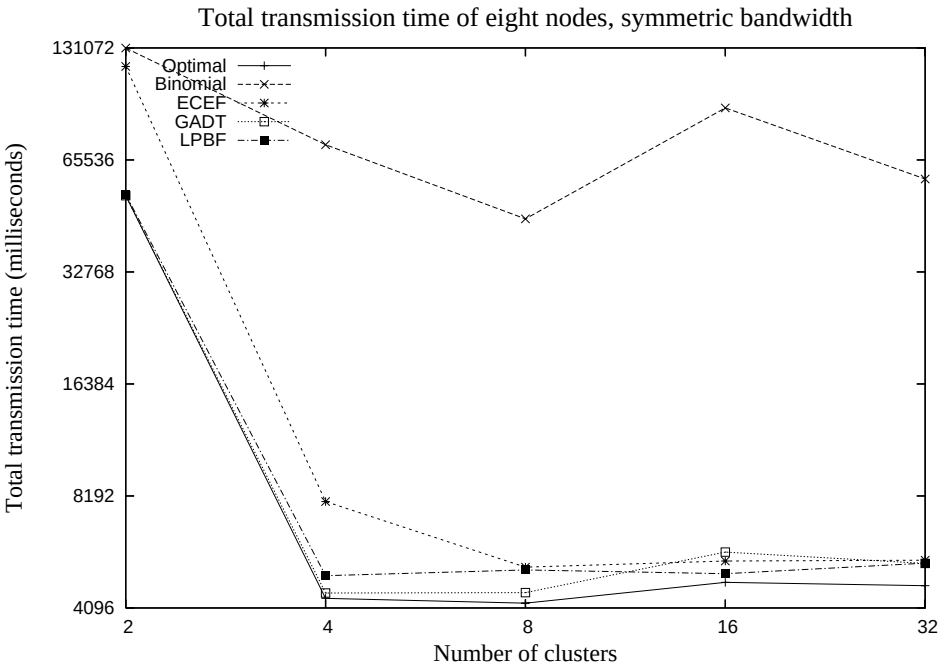

Figure 34 Total transmission time of eight nodes, symmetric bandwidth

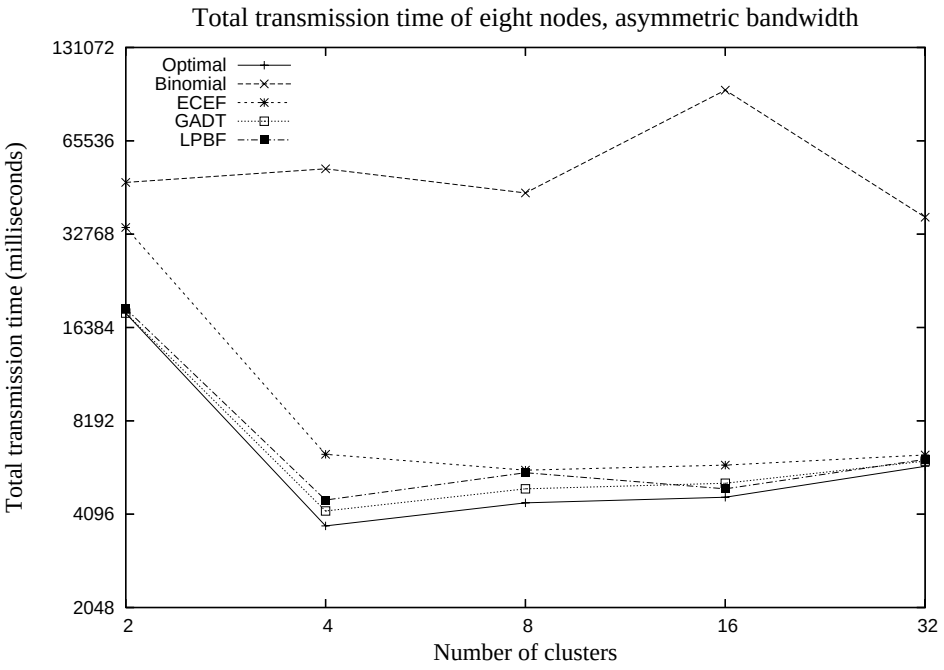

Figure 35 Total transmission time of eight nodes, asymmetric bandwidth

Table 10 **Normalized total transmission time of eight nodes, symmetric bandwidth**

Grid Size	Algorithm			
	Binomial	**ECEF**	**GADT**	**LPBF**
2	2.49	2.22	1.00	1.01
4	16.57	1.82	1.03	1.15
8	10.78	1.25	1.07	1.23
16	18.85	1.14	1.21	1.06
32	12.39	1.17	1.15	1.15
Average	12.22	1.52	1.09	1.12
Standard derivation	6.32	0.48	0.08	0.09

Table 11 **Normalized total transmission time of eight nodes, asymmetric bandwidth**

Grid Size	Algorithm			
	Binomial	**ECEF**	**GADT**	**LPBF**
2	2.64	1.89	1.00	1.03
4	14.17	1.70	1.12	1.21
8	9.99	1.27	1.11	1.25
16	20.56	1.27	1.11	1.06
32	6.36	1.09	1.04	1.05
Average	10.74	1.44	1.08	1.12
Standard Derivation	6.95	0.34	0.05	0.10

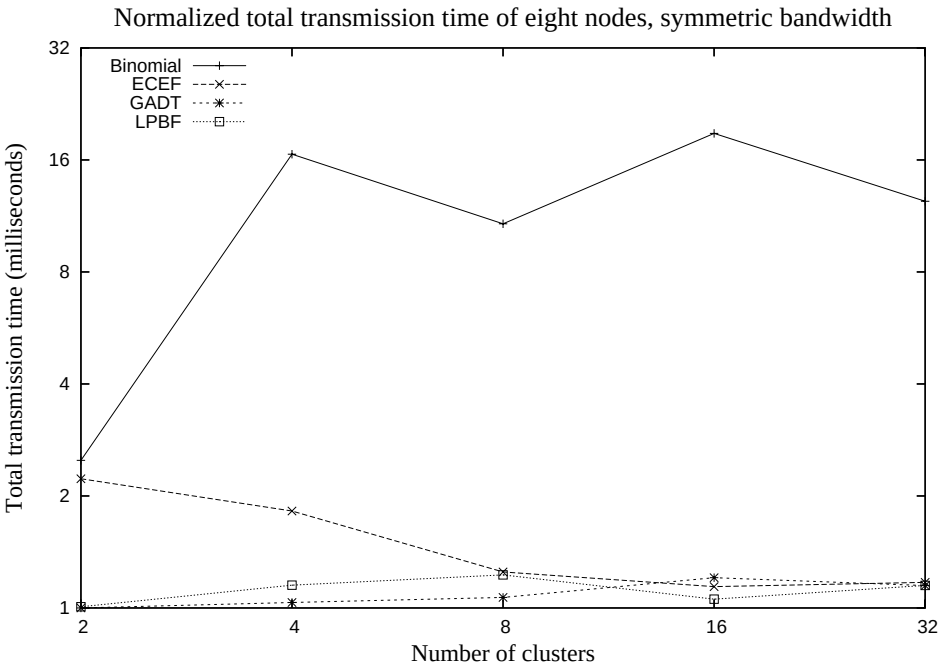

Figure 36 Normalized total transmission time of eight nodes, symmetric bandwidth

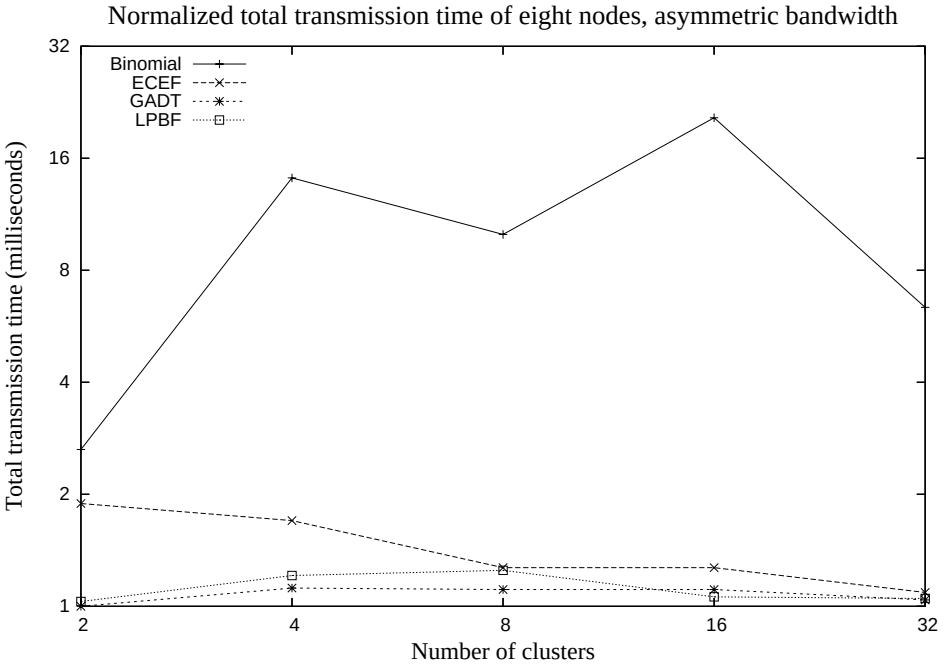

Figure 37 Normalized total transmission time of eight nodes, asymmetric bandwidth

Table 10 and Table 11 show that GADT and LPBF can produce good multicast schedules which are very close to the optimal. When comparing with the

optimal algorithm, GADT which produces a better multicast schedule than LPBF. This is the result of GADT produces a multicast schedule from a number of them and evolves the schedules in order to find the best one. LPBF is a two-level algorithm that minimizes the total transmission time by minimizing intercluster communications and uses the binomial tree algorithm for intracluster communications. LPBF also maximizes the communication overlapping by scheduling a multicast task in the order of the length of branch in the multicast schedule. The results show that LPBF can generate a multicast schedule close to the optimum. The standard derivation of normalized runtimes shows that the generated multicast schedule is low variance. The multicast schedules generated from binomial and ECEF algorithm has lower performance because they are one-level algorithm; they do not minimize intercluster communications. The results also show that the symmetry of a bandwidth matrix does not affect the performance of algorithms.

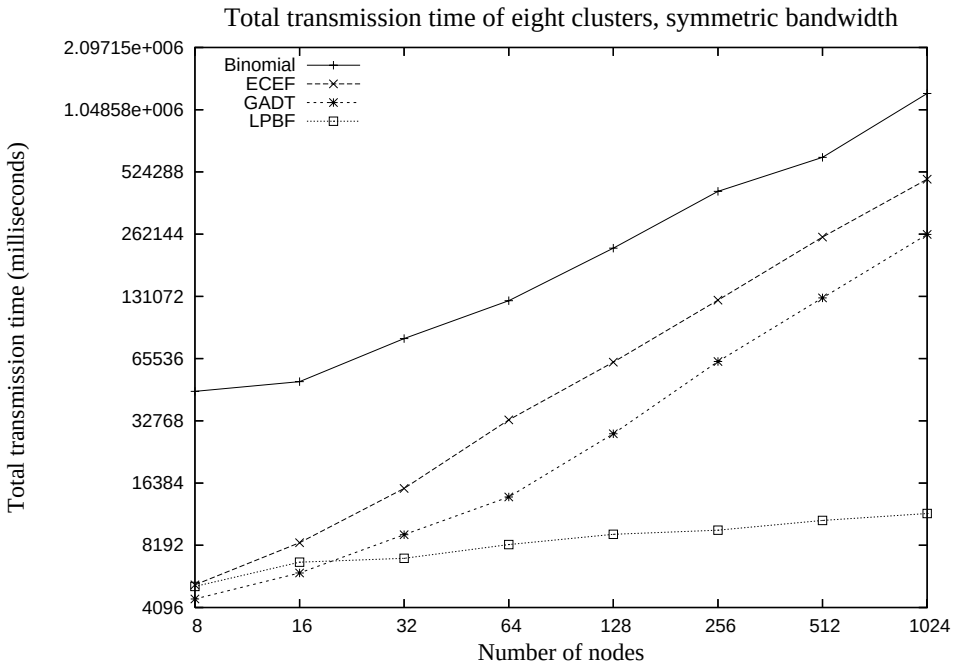

Figure 38 Total transmission time of eight clusters, symmetric bandwidth

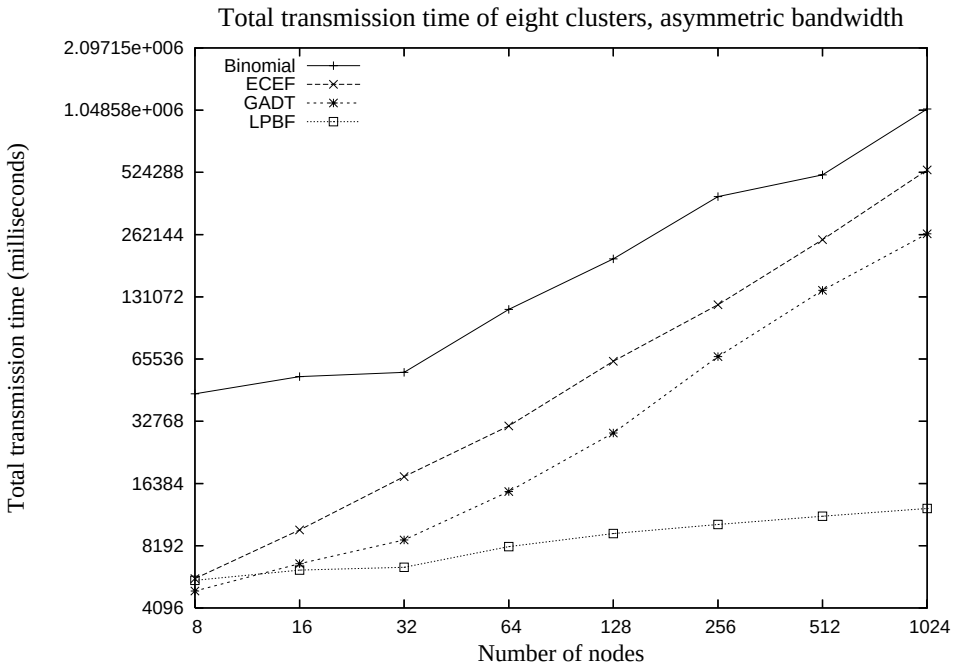

Figure 39 Total transmission time of eight clusters, asymmetric bandwidth

Figure 38 and Figure 39 show the simulated total transmission time when the number of clusters is fixed to eight clusters and the number of nodes varies from 8 to 1024. The optimal algorithm is not included in this test due to a very long computation time. The figures show that when the number of nodes is greater than 16, LPBF generates the best multicast schedule. The total transmission time of a multicast schedule created from LPBF increases slowly because when the number of clusters is fixed, the number of intercluster communication is also fixed; which is eight times. After the intercluster communications finish, the intracluster communication can be done simultaneously. The multicast schedule generated from GADT has a lower performance than LPBF because the algorithm stops at the local maxima. The binomial and ECEF algorithm generates poor performance multicast schedules because they are one-level algorithm; the number of intercluster communications increases with the number of nodes. As stated above, the symmetry of the bandwidth matrix does not affect the performance of algorithms.

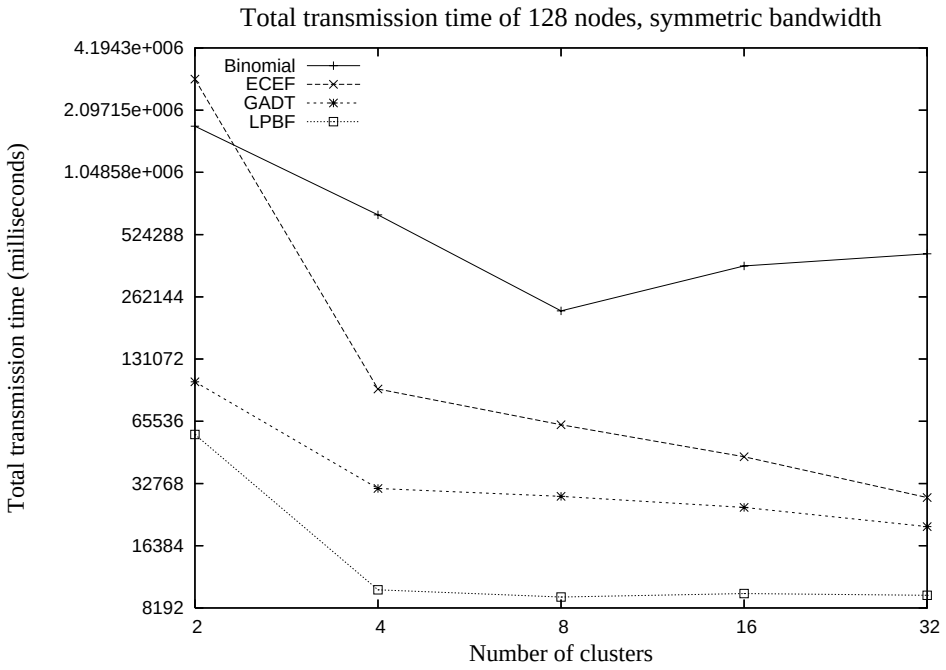

Figure 40 Total transmission time of 128 nodes, symmetric bandwidth

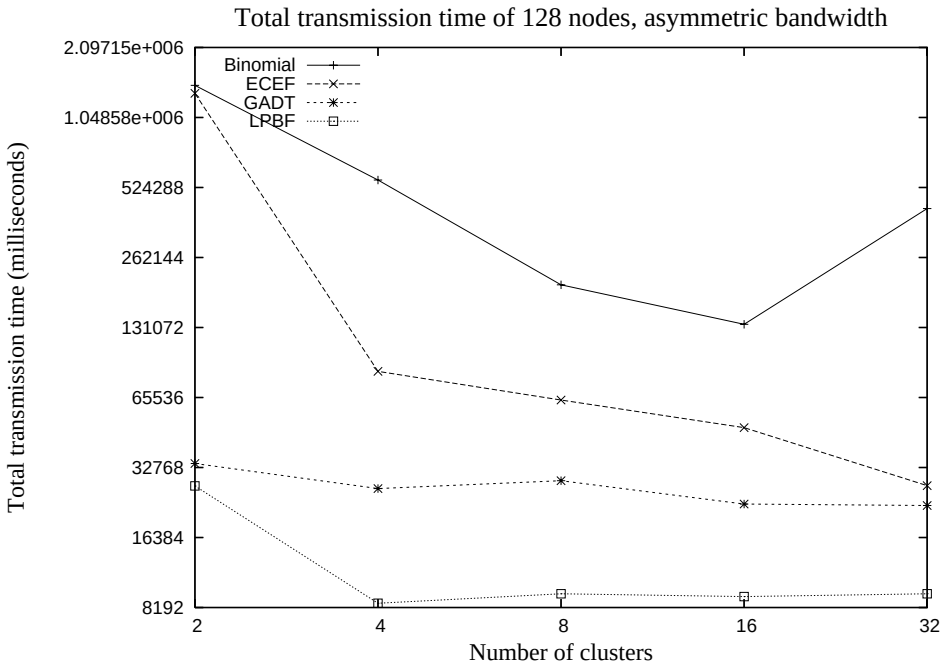

Figure 41 Total transmission time of 128 nodes, asymmetric bandwidth

Figure 40 and Figure 41 show the simulated total transmission time when the number of nodes is fixed to 128 nodes and the number of cluster varies from 2 to 32. Note that, the number of nodes is fixed; the number of multicast task is also fixed at 127. So, when the number of clusters increases, the total transmission time of all

algorithms decreases because the bandwidth sharing between clusters decreases. LPBF is still the best algorithm and GADT is the second. One-level algorithms still produce poor multicast schedule. In the case of two clusters, the total transmission time is high because the generated bandwidth of those two clusters is low.

TreeSim Performances

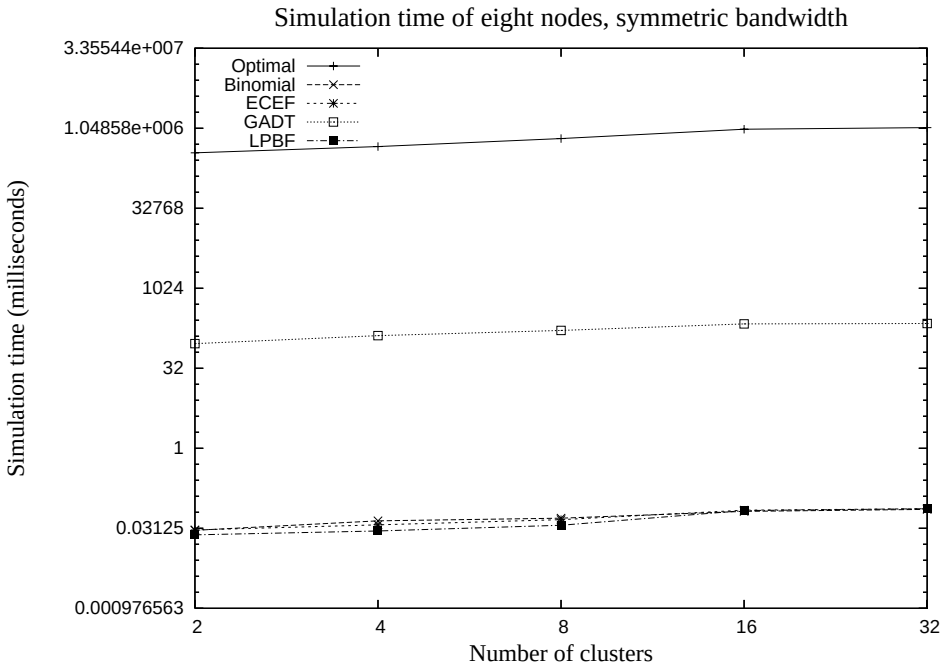

Figure 42 Simulation time for eight nodes, symmetric bandwidth

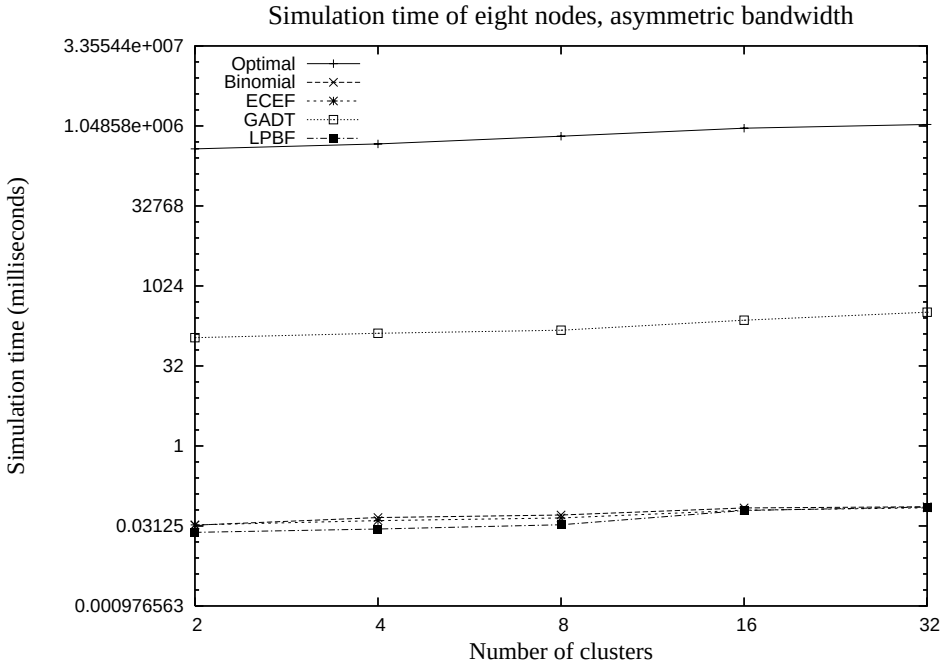

Figure 43 Simulation time for eight nodes, asymmetric bandwidth

Figure 42 and Figure 43 show the simulation time of five algorithms. The simulation consists of algorithm's decision time and schedule's evaluation time. The result shows that the simulation time of optimal algorithms is very high because it uses exhaustive searches on all possible cases. The simulation time of GADT is rather high because it computes a number of multicast schedules to find a near optimal one. The simulation time of heuristic algorithms, binomial, ECEF, and LPBF are low because the complexity of algorithms is very low. The binomial algorithm is *O(log n)* while ECEF and LPBF are $O(n^2)$. However, the simulation time of LPBF is the lowest because the evaluation time dominates an over all simulation time. According to TreeSim's algorithm, the complexity of intercluster tasks simulation is higher than that of intracluster. In case of intercluster tasks, a transmission time cannot be determined immediately; it is maintained in the running task list and this list must be updated every time a new task is added; this resulhs in higher simulation time. In contrast, in the case of intracluster tasks, a transmission time can be determined immediately from intracluster bandwidth and the task is moved to the finish list. So, the number of intercluster communications has the main effect to an overall simulation time. LPBF algorithm minimizes intercluster communications, so, the evaluation time is the lowest. This yields the lowest simulation time. When the number of clusters increases, the simulation time increases because it produces more intercluster communications; that is the result in a longer simulation time. The symmetry of the bandwidth matrix does not affect simulation times because the property does not affect the number of intercluster/intracluster communications.

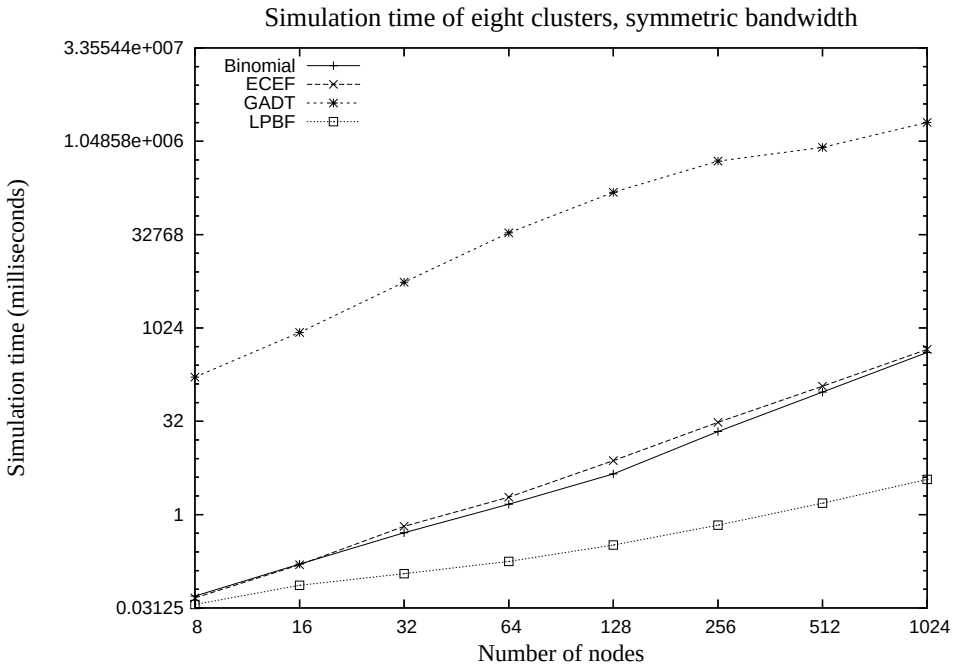

Figure 44 Simulation time for eight clusters, symmetric bandwidth

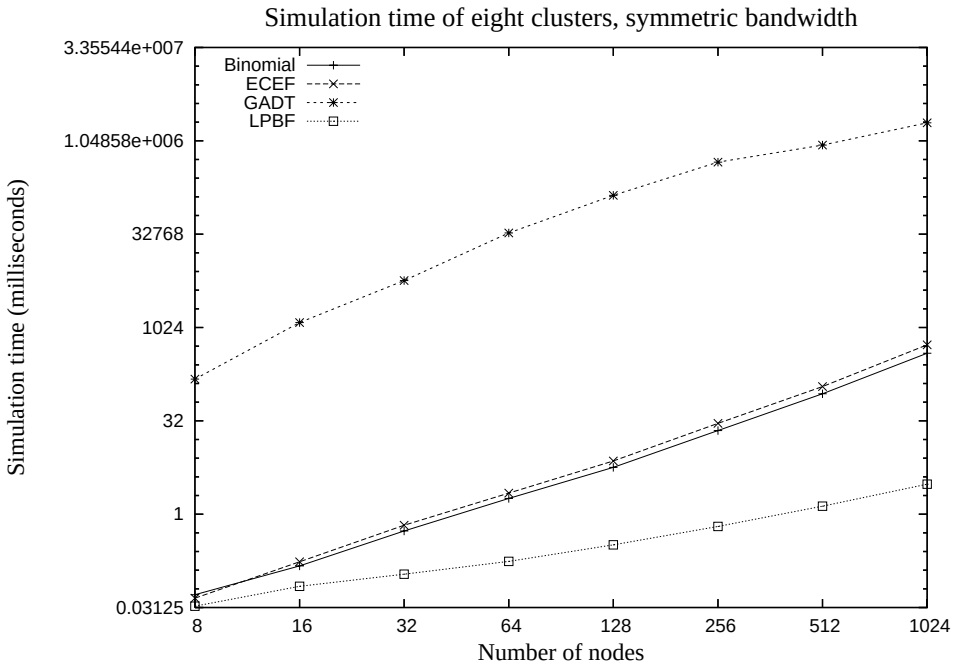

Figure 45 Simulation time for eight nodes, asymmetric bandwidth

Figure 44 and Figure 45 show the simulation time of binomial, ECEF, GADT, and LPBF algorithms when the number of clusters is fixed to eight and the number of nodes varies from 8 to 1024. The figures show that when the number of nodes increases, the simulation time increases linearly as the function of node. The

simulation time of GADT is very long because TreeSim must evaluate all instances created from GADT. The simulation time of binomial and ECEF algorithm are high because they are one-level algorithms; there are many intercluster communications. The simulation time of LPBF is the lowest because it minimizes the intercluster communication and most of multicast tasks are intracluster communications that can be computed a transmission of the task immediately.

Figure 46 and Figure 47 show the simulation time of binomial, ECEF, GADT, and LPBF algorithm when the number of nodes is fixed to 128. The figures show that when the number of clusters increases, the simulation time increases. This is a result of an increasing of the number of intercluster communications. The results show that the simulation time depends mainly on the number of nodes.

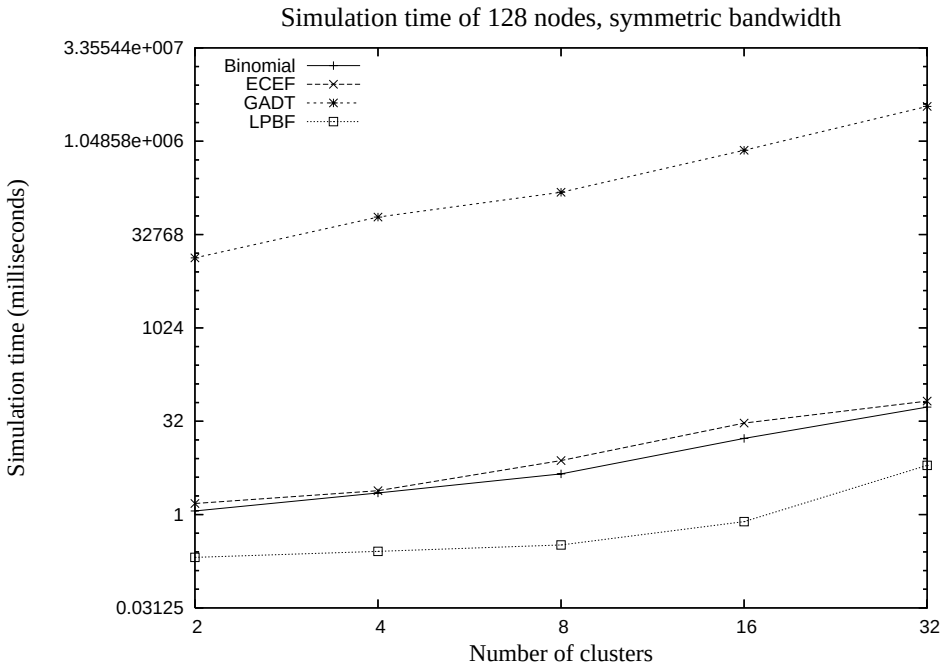

Figure 46 Simulation time for 128 nodes, symmetric bandwidth

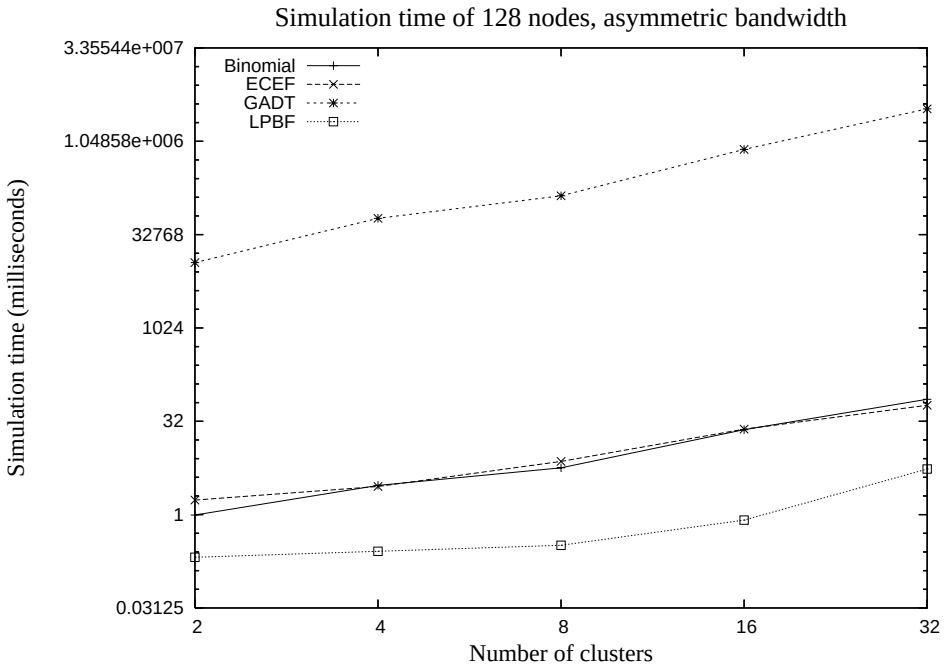

Figure 47 Simulation time for 128 nodes, asymmetric bandwidth

MPITH Performances

The prototype implementation is evaluated on clusters as shown in Table 12. The environment is a part of PRAGMA. Figure 48 shows the topology of the test bed.

Table 12 A test bed configuration

Cluster	Type	Nodes	Brief Configuration	Network
Maeka	Close	32	Opteron 244, 3GB RAM	Gigabit Ethernet
Magi	Open	4	Athlon XP 2400+, 512 MB RAM	Fast Ethernet
Amata	Close	15	Athlon 1 GHz, 512 MB RAM	Fast Ethernet
Aida's Blade	Close	17	Dual Pentium III 1.4 GHz, 512 RAM	Fast Ethernet
Rocks-52	Close/Open	32	Quad Intel XEON 3 GHz, 4 GB RAM	Gigabit Ethernet

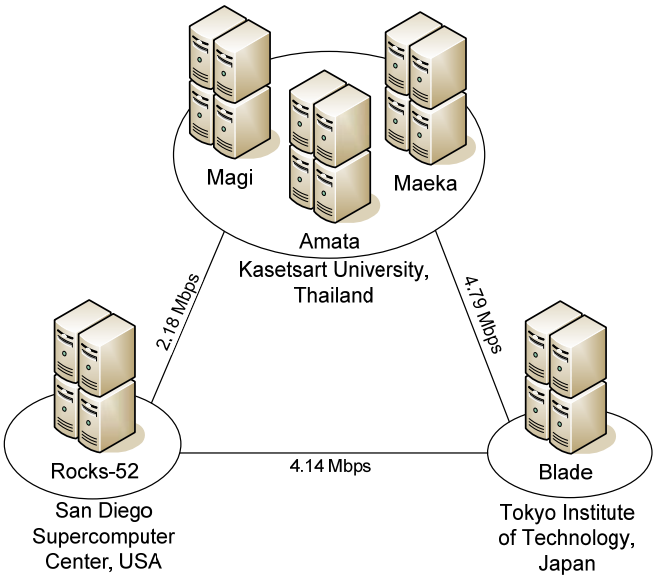

Figure 48 **The test bed topology**

MPITH is evaluated in three areas: point-to-point, broadcast, and application performance in comparison with MPICH2 1.0.2. All tests presented here were done ten times and the average values are reported.

1. Point to Point

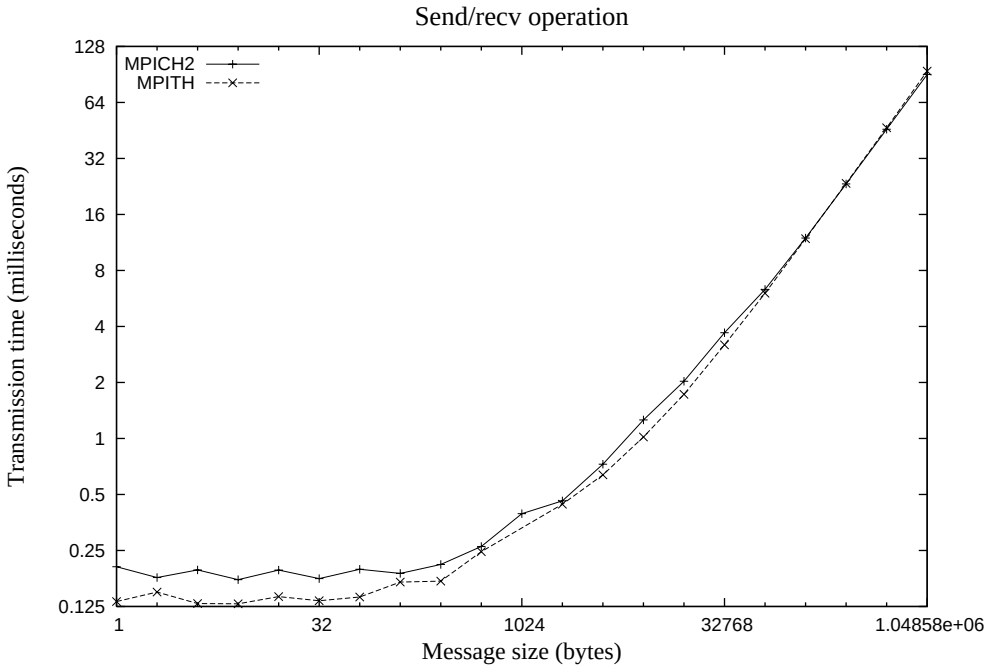

Figure 49 **A performance of MPI_Send/MPI_Recv on Magi cluster**

Figure 49 shows the comparison of MPI_Send/MPI_Recv operation between MPITH and MPICH2 on the Magi cluster. The result shows that, for small messages, MPITH performance is better than MPICH2, especially for a message size between 1 byte and 64 kB. This is the result of a simpler software stack in MPITH. MPITH always sends messages regardless of message size and a matched receiving call. This mechanism has the advantage of fast message matched at the receiving side. For large message size, both MPITH and MPICH2 exhibit similar levels of performance. For a very large message size, more than two Megabytes, MPICH2 should exhibit a better performance because the message is transmitted to the user's buffer directly without any buffering. This performance penalty is in an order of memory bandwidth of receiving side. However, in a Grid system, an intercluster bandwidth is low when compared to the memory bandwidth, so, an intercluster transmission time dominates overall send/receive time. MPITH buffering policy is still useful in a Grid system.

In NAT environment, MPITH contains a built-in message forwarder, which is for forwarding messages from external to internal and vice versa. Figure 50 shows the comparison between a single-hop and double-hops communications. The single-hop is a communication between the head node of Amata and Rocks-52 cluster, while the double-hops is a communication between the head node of Amata and a compute node of Rocks-52 cluster. The figure shows that over a Grid system, the difference between single- and double-hop of MPI communication time is very low. This is the result of network bandwidth in the Grid system, which is low in this test environment compared to local network. The intercluster transmission time dominates an overall send/receive operation. Moreover, the bandwidth between these two clusters measured by Iperf (Ajay, 2005) is 2.4 Mbit/s, this shows that MPITH's message forwarder routine performs very well.

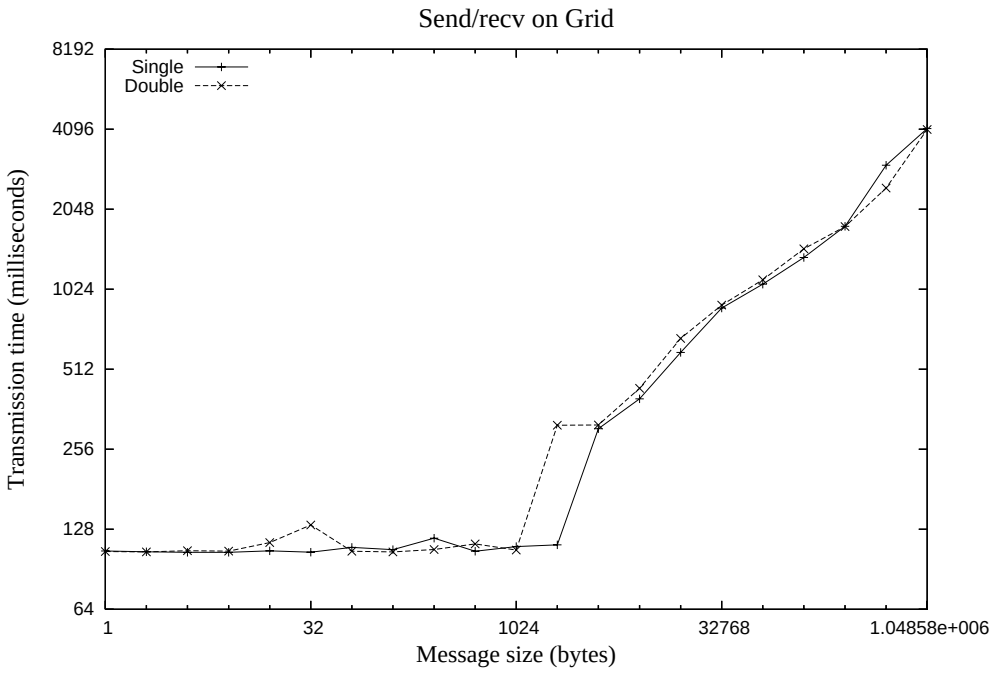

Figure 50 A comparison between single and double hops MPI_Send/MPI_Recv

2. Broadcast

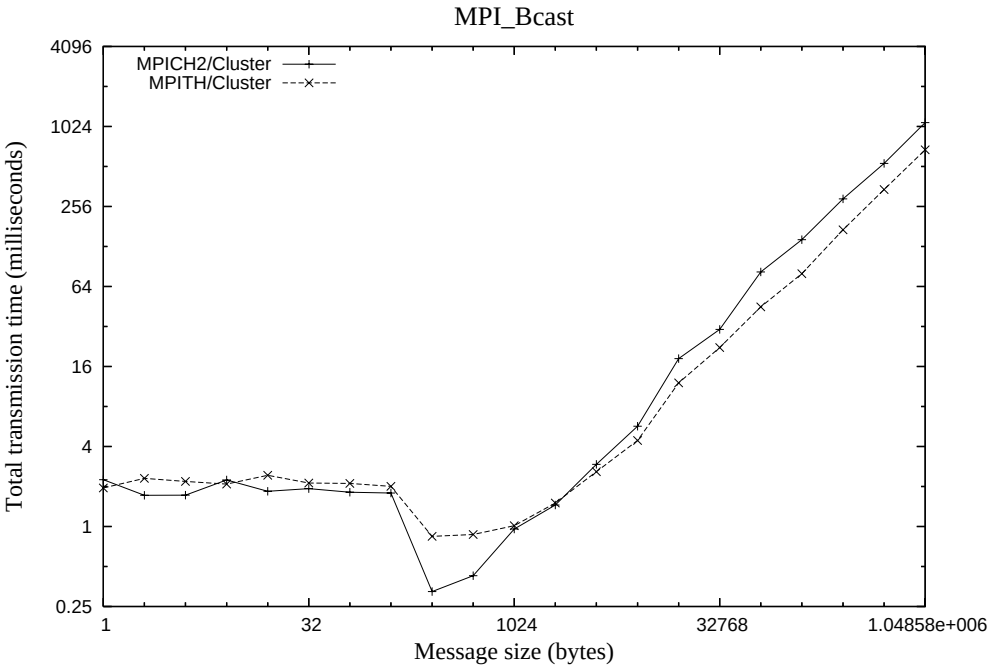

Figure 51 A performance of MPI_Bcast in a cluster environment

Figure 51 shows the comparison between MPICH2 and MPITH in cluster. The 24 nodes in Maeka cluster are the test bed for the cluster environment. The result shows that, for a small message size, the total transmission time depends mainly on latency that comes from the Linux buffer policy. When the message is smaller than about 1 kB, messages fit in the Linux kernel buffer, hence Linux waits until the buffer is full. For a large message size, the message is larger than the buffer, hence Linux flushes a message immediately. The figure shows that MPITH is slightly better than MPICH2 in a cluster environment.

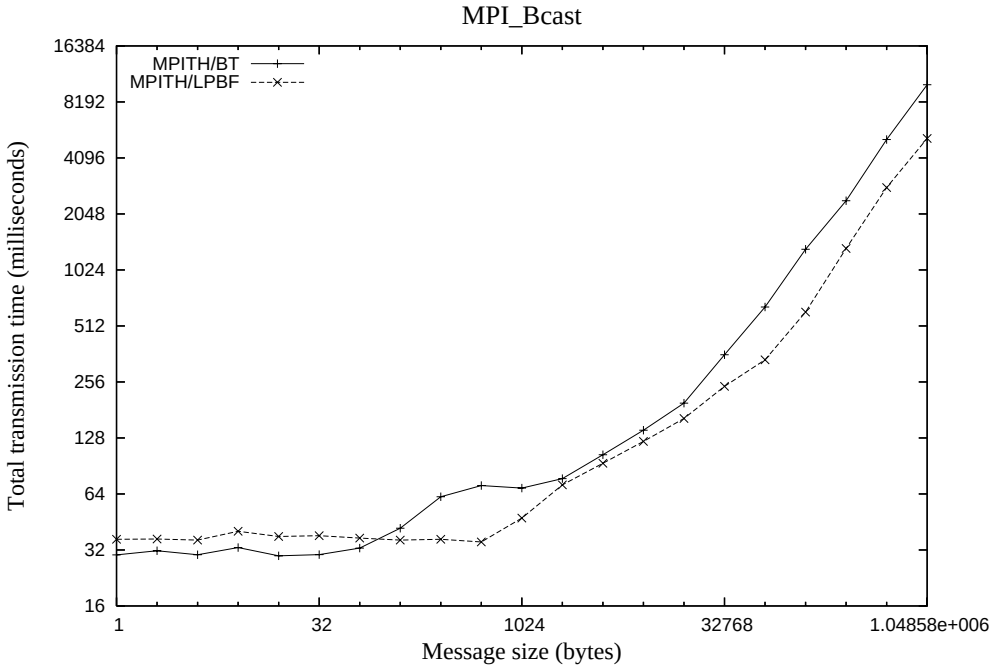

Figure 52 A performance of MPI_Bcast in a Grid environment

Figure 52 shows the comparison between two MPITH implementations that use traditional binomial tree and LPBF algorithm in a Grid system. Eight nodes from Amata, Rocks-52, and Blade cluster are the test bed for Grid environment. All clusters are close clusters. A message transmission from any compute node must be forwarded by a front-end node. From the figure, when message size is less than 64 bytes, the binomial tree algorithm performs slightly better than LPBF. In the binomial algorithm, a number of messages are sent to a front-end node simultaneously. Messages are quickly flushed. This benefits an overall performance. Although, the number of intercluster communications is high, the intercluster communication time is small and the front-end node can forward small messages without any congestion. For a large message size, the intercluster communication time increases significantly. LPBF that takes a network topology into account performs better than the binomial algorithm.

3. Reduce

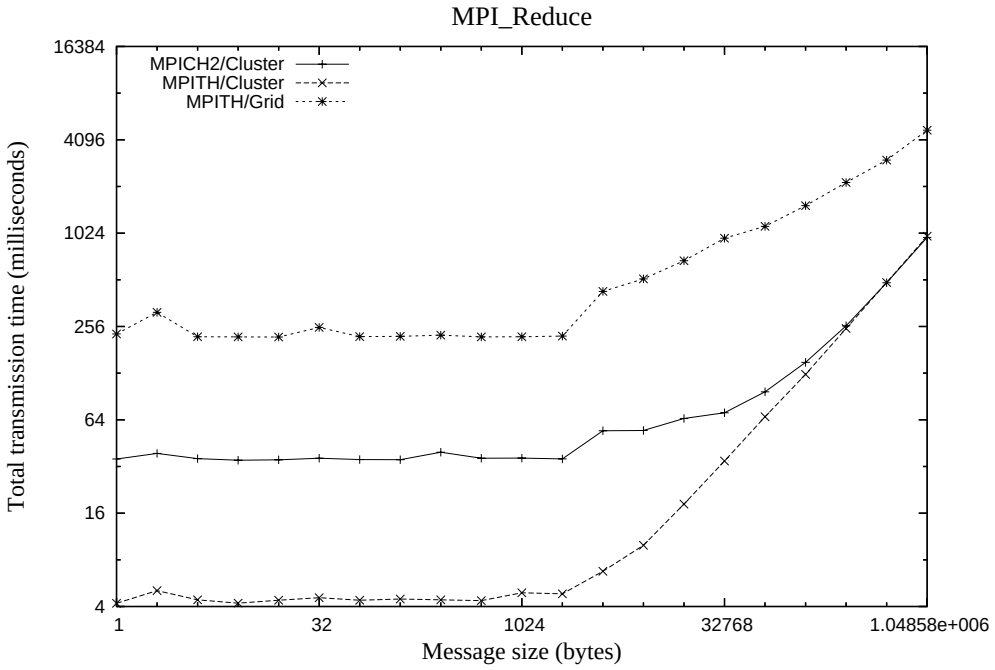

Figure 53 A performance of MPI_Reduce

Figure 53 shows the performance of MPI_Reduce. The lines labeled MPICH2/Cluster and MPITH/Cluster are results when testing using 24 nodes of Maeka cluster. The labeled MPITH/Grid lines are results when testing using eight nodes from Amata, Blade, and Rocks-52 cluster. The graph shows that in a cluster environment, MPITH has a better performance compared to that of MPICH2 when messages are less than 32 kB. When message sizes are larger than 32 kB, MPITH has the same level of performance compared to MPICH2. The implementation of MPI_Reduce is an inverse of MPI_Bcast, plus an additional mathematical operation. So, MPITH performs better than MPICH2 because an implementation of MPITH's MPI_Bcast is better. In a Grid environment, the performance is lower than one in a cluster environment because of the higher network latency and lower bandwidth. However, the difference of total transmission time between Grid's and cluster's is a constant value.

4. Scatter

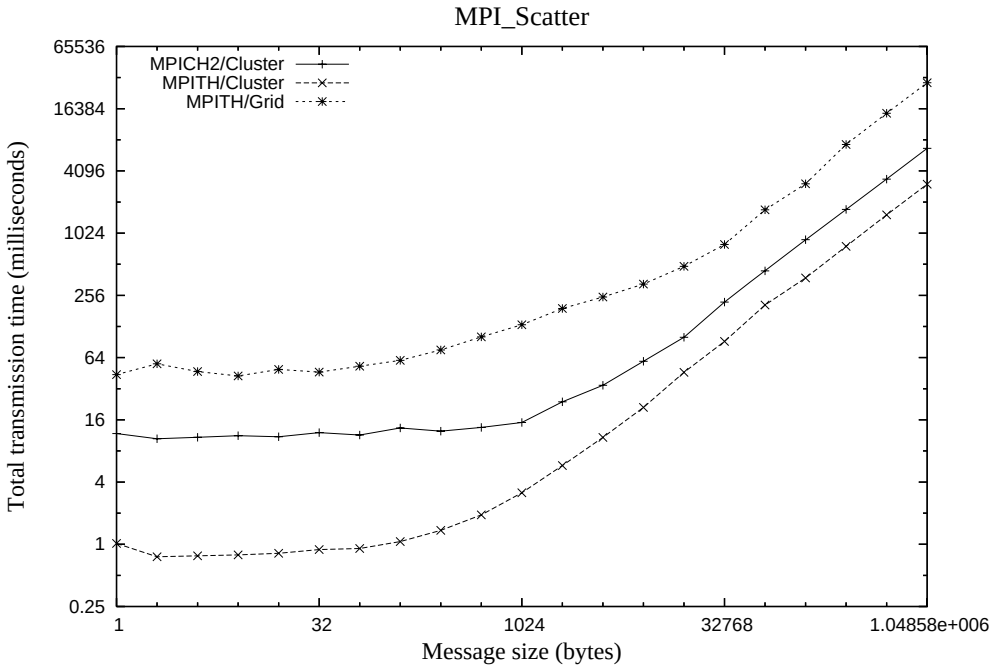

Figure 54 A performance of MPI_Scatter

Figure 54 shows the performance of MPI_Scatter. The test environment is the same as that of MPI_Reduce. The results show that, in a cluster environment, MPITH has a better performance than that of MPICH2. This is a result of a simpler software stack in MPITH. In a Grid environment, the total transmission time is longer than that of a cluster environment because of a lower bandwidth and a higher network latency.

5. Gather

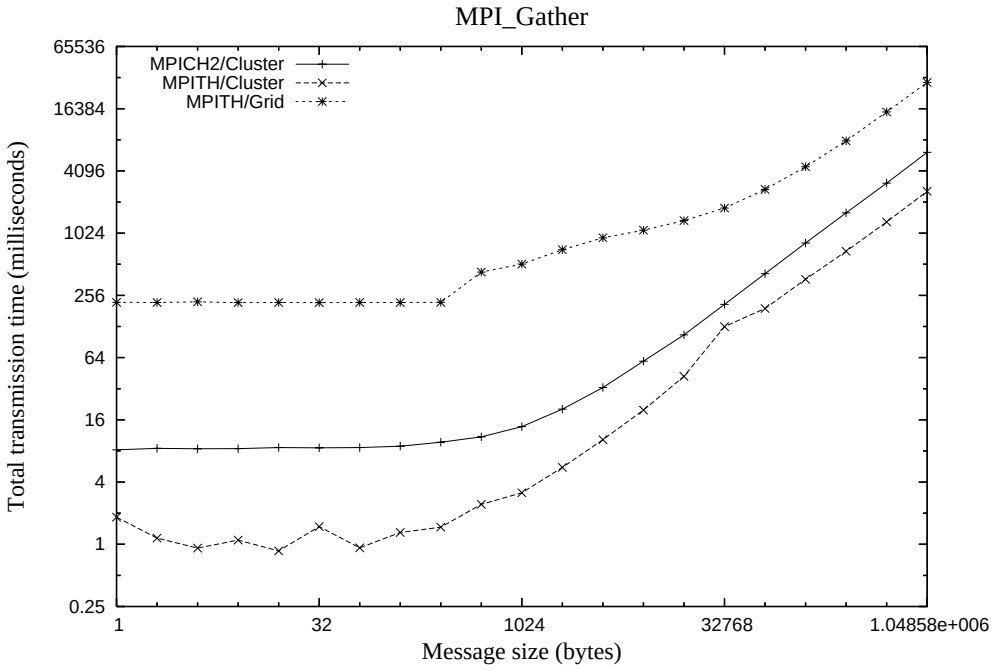

Figure 55 A performance of MPI_Gather

Figure 55 shows the performance of MPI_Gather. The test environment is the same as that of MPI_Reduce. The results show that, in a cluster environment, when a message is smaller than 32 kB, MPITH has a better performance than that of MPICH2. When a message is larger than 32 kB, both MPITH and MPICH2 have the same level of performance. This is the same reason as MPI_Scatter because MPI_Gather is an inverse of MPI_Scatter. In a Grid environment, the total transmission time is longer than that of a cluster environment because of a lower bandwidth and higher network latency.

6. Application

There are three applications, namely, heat flow, Gaussian elimination, and VaR application. Table 13 shows brief information about these applications. 24 nodes of Maeka cluster are the environment of cluster system and eight nodes of Amata, Rocks-52, and Blade cluster are the test bed for Grid environment. Note that CPUs of Maeka are faster than Amata's and Blade's. So, a parallel application performance of a Grid environment is slower than that of Maeka cluster especially when the number of nodes equals. The sequential implementations of these applications are tested using a compute node in the Amata cluster.

Table 13 Test applications

Application	Communication volume
Heat flow	High
Gaussian elimination	Medium
VaR	Low

The first application is incompressible heat flow application or heat flow. Heat flow is a fluid dynamic application that computes heat transfer in electronics equipment, which can simulate laminar and turbulent of incompressible heat transfer. The application runs iteratively until an error is less than a predefined value that is 10^{-10} in this test. The application runs approximately 1500 iterations on a problem size of 169×169. Figure 56 and Table 14 show the running time and speed up of the application, respectively. The results show that the runtime of MPITH and MPICH2 is the same. A parallel implementation of this program uses MPI_Send and MPI_Recv function to exchange data with neighbor cells. According to MPI_Send/MPI_Recv performance, MPITH and MPICH2 have the same level of performance. So, under a cluster environment, the running time of MPITH and MPICH2 are the same. Table 15 shows a normalized running time that uses MPICH2 as a based line. The runtime in Grid system is very high, which is 50 times slower than in cluster environment. This is a result of fine grain behavior of the heat flow application. The speedup in case of a Grid environment, which is less than one, shows that the high communication volume still needs a cluster.

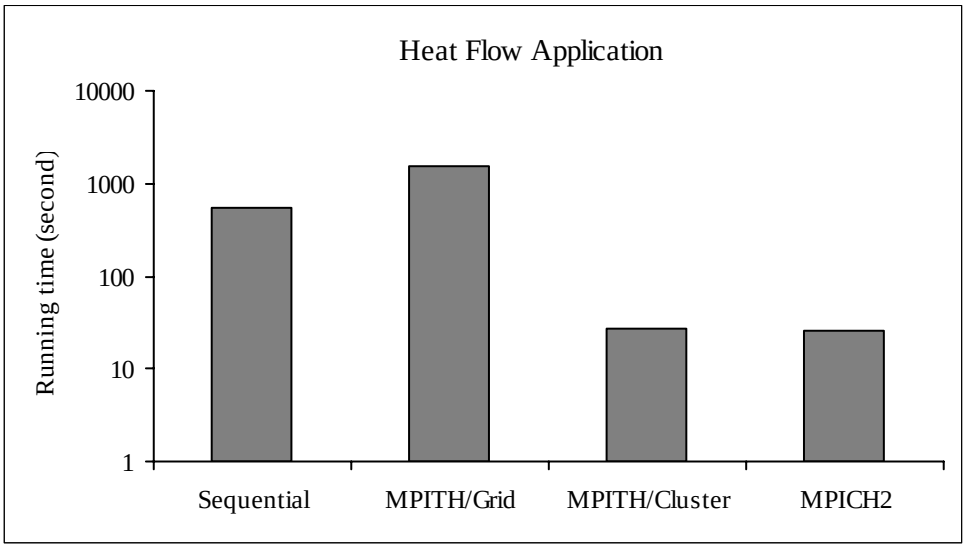

Figure 56 Runtime of heat flow application

Table 14 Speedup of heat transfer application

Environment	Normalized Runtime
MPITH/Grid	0.35
MPITH/Cluster	20.10
MPICH2/Cluster	20.77

Table 15 Normalized runtime of heat transfer application

Environment	Normalized Runtime
MPITH/Grid	59.21
MPITH/Cluster	1.03
MPICH2/Cluster	1.00
Sequential	20.77

Next, the runtime of parallel Gaussian elimination application is evaluated. The application contains less communication volume than heat flow application. Figure 57 and Figure 58 show runtime and speed up of the application. From figures, the runtimes in a cluster environment of both MPITH and MPICH2 are the same and about four to ten times speedup. The parallel implementation of Gaussian elimination application uses MPI_Bcast, MPI_Scatter, and MPI_Gather. According to the performance of these functions, MPITH and MPICH2 have the same performance level. So, under a cluster environment, the runtimes of the application are very close. The runtime in a Grid environment is slower than a sequential program when the problem size is smaller than 2800 because a communication time dominates an overall application running time. When the problem size is bigger, the performance is better than that of sequential because a computation time dominates an overall application running time. This means that a moderate communication application with a big enough problem size can be run on a Grid system. Figure 59 shows the normalized runtime of Gaussian Elimination based-on the runtime of MPICH2 in a cluster environment which shows that the runtime over Grid system is approximately seven times slower than that of a cluster system. This is a result of high latency/low bandwidth of WAN link between clusters and CPU speed in a Grid environment.

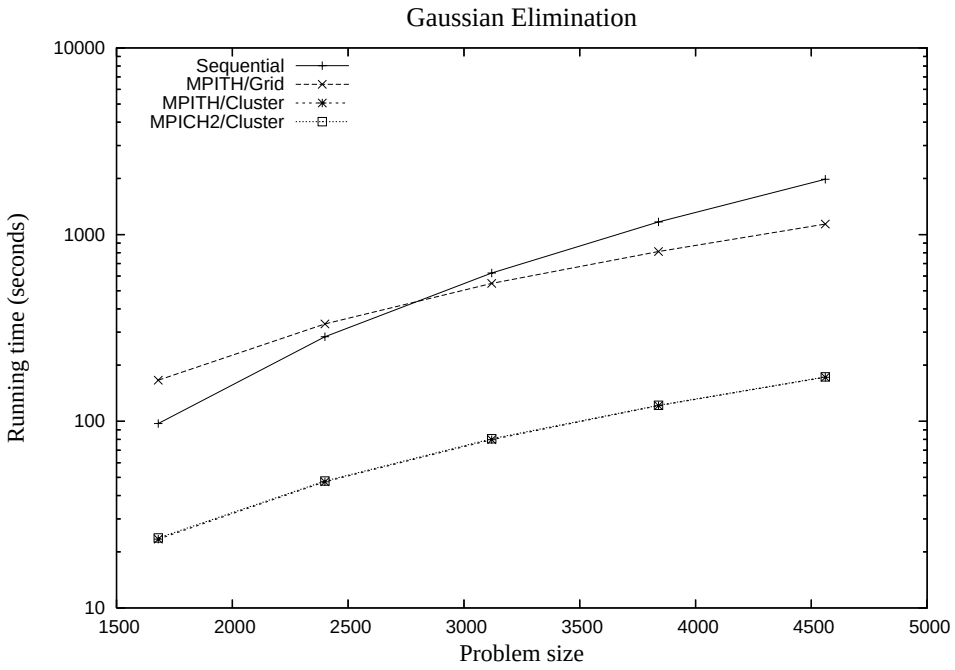

Figure 57 **Runtime of Gaussian elimination**

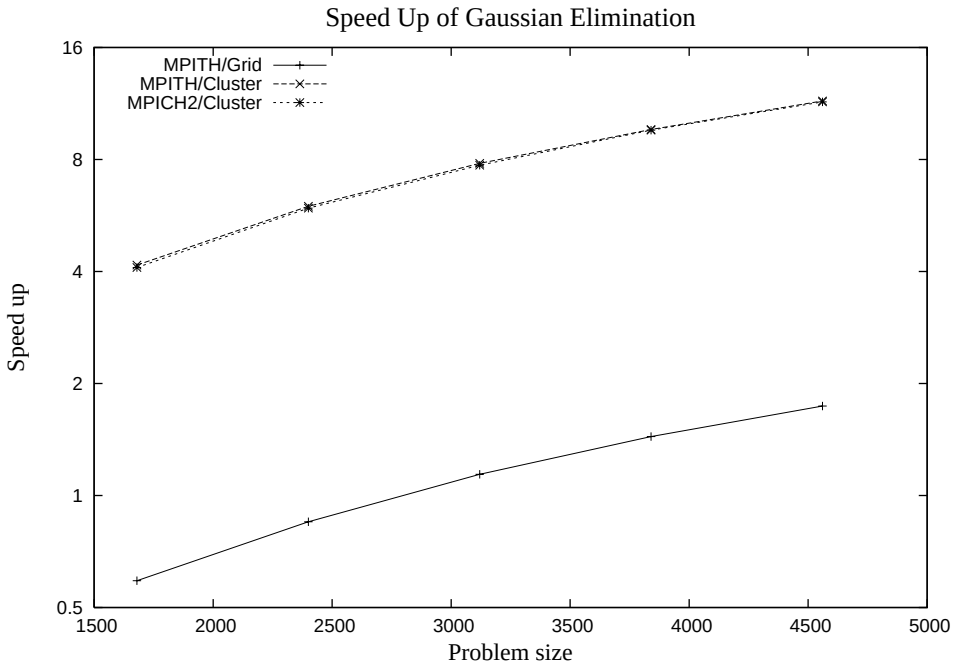

Figure 58 **Speed up of Gaussian elimination**

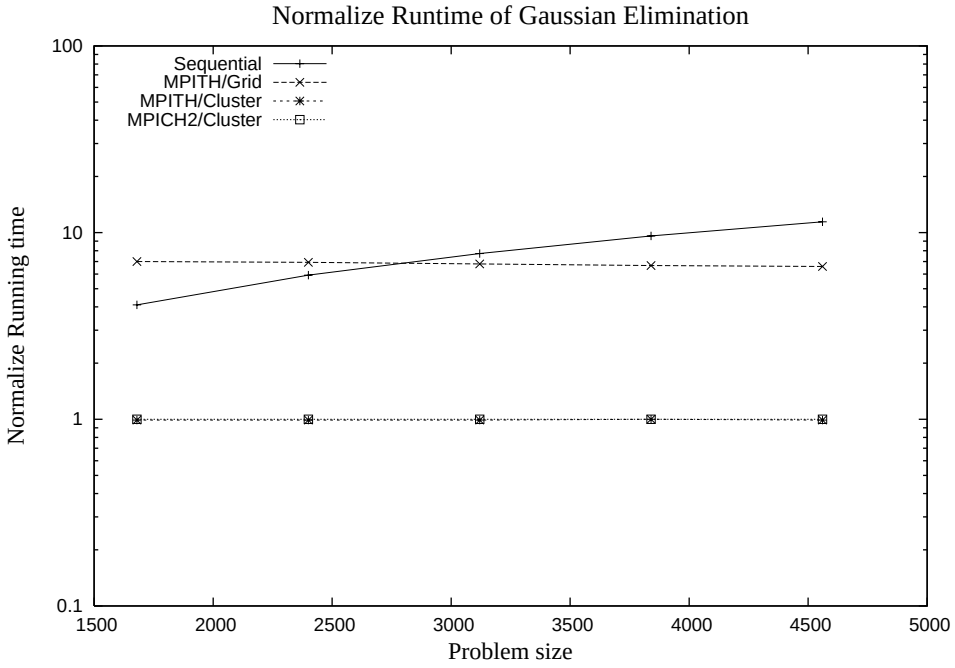

Figure 59 Normalized runtime of Gaussian elimination

Value at Risk (VaR), the last tested application, is a financial application that predicts the cost of bond, and stock in the future. It is an embarrassingly parallel application. Figure 60 shows that, in a cluster environment, both MPITH and MPICH2 perform the same level of performance. Figure 61 shows the speed up of VaR application. It shows that performance of the application in Grid is about 20 times, which is near the perfect speed up. The speed up of cluster environment is super linear speed up because Maeka machine is faster than Amata, which is used as a based case. The speed up of Grid environment, which is more than one, shows that a Grid environment can be used to run a low communication application. Figure 62 shows the normalized runtime based on MPICH2. The figure shows that the performance in the Grid system is slower than that of a cluster about two times when compared with the same number of node. This means that, if more nodes in Grid system are utilized, the performance of this class of application is better than that of a cluster environment.

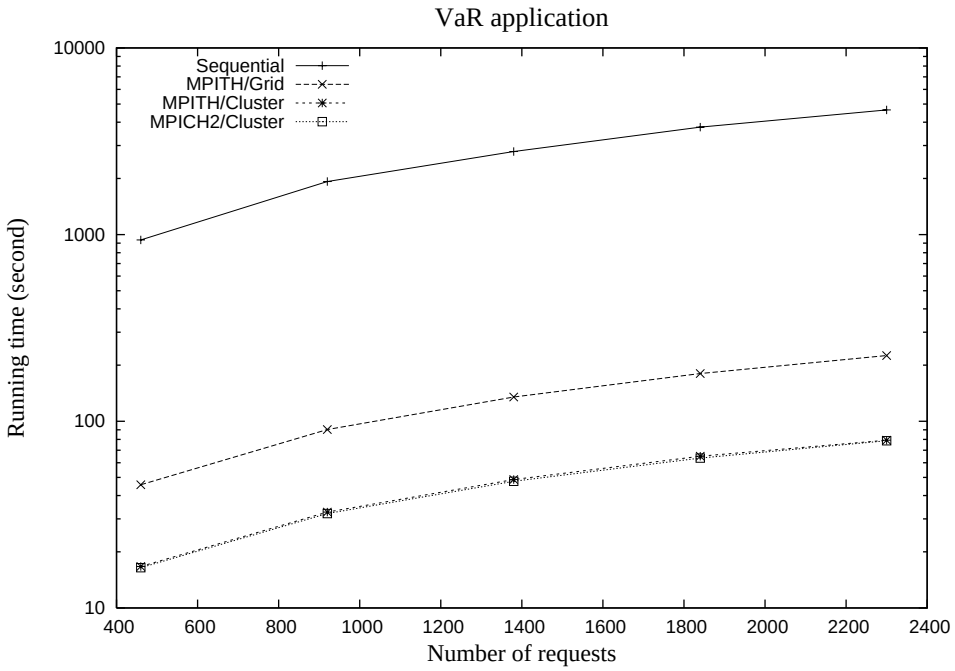

Figure 60 **Runtime of VaR application**

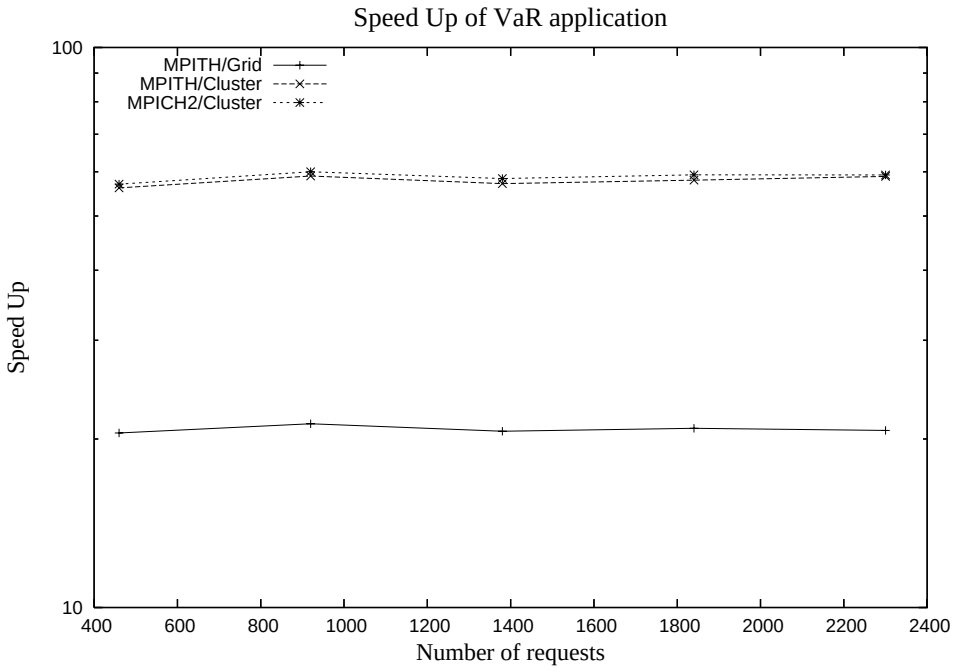

Figure 61 **Speed up of VaR application**

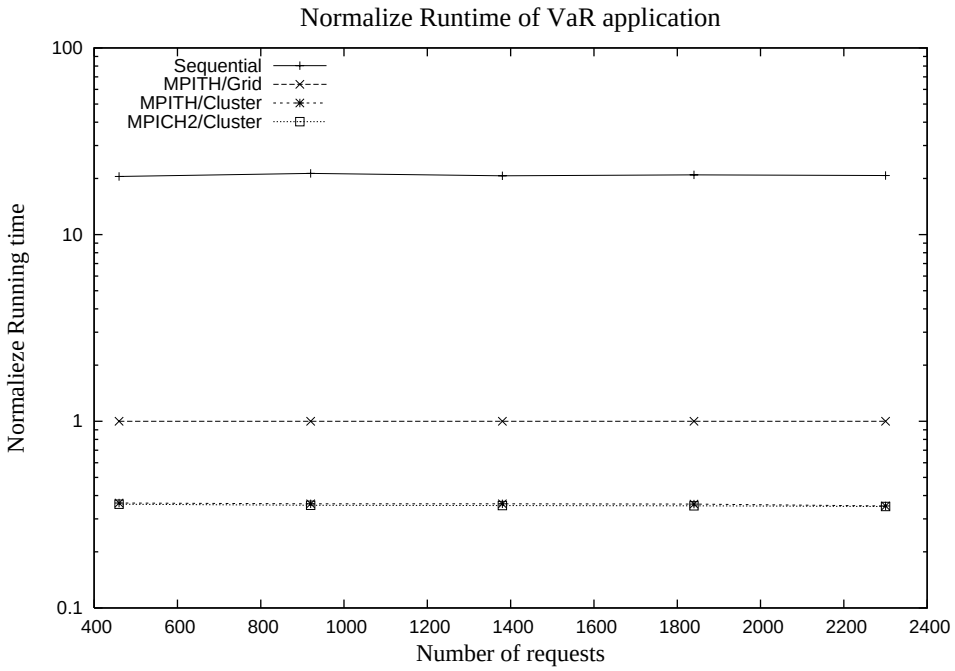

Figure 62 Normalized runtimes of VaR application

CONCLUSION

This dissertation addresses two challenges in a Grid system that are a closed cluster environment and an efficient multicast communication algorithm. For the first problem, this dissertation proposes a systematic model of routing algorithm based on a virtual cluster model or VC. The VC models logical network connections as an undirected graph called a direct connectivity graph or DCG. Then, a virtual cluster graph is formed and routes are created from it. The HVC provides an efficient methodology for creating a routing table for a Grid system.

For the second problem, this dissertation proposes two algorithms. GADT, the first algorithm, is an off-line algorithm used for creating a near optimal multicast algorithm when the number of nodes is small. LPBF, the second algorithm, is designed for an online algorithm used in MPI runtime library. The simulation shows that LPBF can generate an efficient multicast algorithm because it minimizes intercluster communication and maximizes the communication overlapping among multiple branches in a multicast tree.

The proposed algorithms are implemented in a prototype MPI library named MPITH. MPITH complies with a subset of mostly used MPI functions. It supports a Grid by interfacing to Globus for an intercluster remote task execution and synchronization. MPITH integrates a routing and forwarding routine to an MPITH process. So, an extra forwarding daemon is not required. The result shows that MPITH can forward MPI messages with very low performance penalty. MPITH's multicast communication is LPBF algorithm. The results show that the performance of MPITH's multicast communication is significantly improved when compared to that of the binomial algorithm. The application performance shows that the application can utilize nodes in Grid even though they are placed behind a firewall or in a NAT environment.

Possible future works includes several topics. First, a performance model of PCC, for instance, MPI_Scatter and MPI_Gather, may be addressed. A PCC differs from NPCC, for example, MPI_Bcast, in that a message size is changed every time it is forwarded. This can be handled by adding a message size to a multicast task. Next, all-to-all communications both NPCC and PCC can be addressed. This can be done by extending a multicast topology to a general graph. Next, the accuracy of performance prediction can be improved by adding other network parameters, for example, latency, send/receive overhead. Next, GADT can be improved in order to create a near optimal multicast schedule when there are a number of nodes. Currently, GADT calculates a solution from node level. GADT can be adapted to calculate a solution for intercluster communication and use the binomial tree algorithm for intracluster communication. Finally, the implementation of a forwarder can be changed to a nonshared, nonrunnable forwarder in order to decrease resource consumption on a front-end node. A nonshared, nonrunnable forwarder can be implemented by excluding master processes from MPI_COMM_WORLD; leaving the master processes responsible only for forwarding MPI messages.

LITERATURE CITED

Alexandrov, A., M.F. Ionescu, K.E. Schauser and C. Scheiman. 1995. LogGP: Incorporating long messages into the LogP model--one step closer towards a realistic model for parallel computation, pp. 95-105. In **The Seventh Annual ACM Symposium on Parallel Algorithms and Architectures** ACM Press New York, NY, USA.

Aumage, O. and G. Mercier. 2003. MPICH/MADIII: a cluster of clusters enabled MPI implementation, pp. 26-33. In **The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid.**

Bal, H., A. Plaat, M. Bakker, P. Dozy and R. Hofman. 1998. Optimizing parallel applications for wide-area clusters, pp. 784-790. In **The International Parallel Processing Symposium.**

Banikazemi, M., V. Moorthy and D. Panda. 1998. Efficient collective communication on heterogeneous networks of workstations, pp. 460-467. In **International Conference on Parallel Processing.**

Beaumont, O., L. Marchal and Y. Robert . 2005. Broadcast trees for heterogeneous platforms , pp. 80b. In **The 19th IEE International Symposium on Parallel and Distributed Processing.**

Bernaschi, M. and G. Iannello. 1998. Collective communication operations: experimental results vs. theory. **Concurrency - Practice and Experience**. 10(5): 359-386.

Bhat, P.B., C.S. Raghavendra and V.K. Prasanna. 2003. Efficient collective communication in distributed heterogeneous systems. **Journal of Parallel and Distributed Computing**. 63(3): 251-263.

Borella, M., D. Grabelsky, J. Lo and K. Taniguchi. 2001. Realm Specific IP: Protocol Specification

Brune, M.A., G.E. Fagg and M.M. Resch. 1999. Message-passing environments for metacomputing. **Future Generation Computer Systems**. 15(5-6): 699-712.

Burns, G., R. Daoud and J. Vaigl. 1994. LAM: An open cluster environment for MPI, pp. 379-386, In **The 1994 ACM/IEEE Conference on Supercomputing.**

Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein. 2001. **Introduction to Algorithms.** 2nd edition. MPI Press. UK

Goldberg, D.E. 1989, **Genetic Algorithms in Search, Optimization and Machine Learning**, Kluwer Academic Publishers, Boston, MA.

Culler, D., R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T.V. Eicken. 1993. LogP: Towards a realistic model of parallel computation, pp. 1-12. In **The fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. ACM Press New York, NY, USA .

Das, D., R. Sabharwal, S. Saraswati, P.N. Anantharaman and J. Oh. 2005. A network architecture for enabling execution of MPI applications on the Grid**. International Journal of Information Technology.** 11(4):.

Faraj, A. and X. Yuan. 2005. Automatic generation and tuning of MPI collective communication routines, pp. 393-402. In **The 19th Annual International Conference on Supercomputing.** ACM Press New York, NY, USA.

Foster, I. and C. Kesselman . 1997. Globus: A metacomputing infrastructure toolkit. **The International Journal of Supercomputer Applications and High Performance Computing**. 11 (2): 115-128.

_____. and _____. 2003. **The Grid 2: Blueprint for a New Computing Infrastructure**. Morgan Kaufmann.

_____, _____, J. Nick, and S. Tuecke. 2002. The Physiology of the Grid: An open grid services architecture for distributed systems integration. Globus Project.

Gabriel, E., G.E. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham and T.S. Woodall. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. **Recent Advances in Parallel Virtual Machine and Message Passing Interface: the 11th European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science**. 3241: 97-104.

Gropp, W. 2005. MPICH2: A new start for MPI implementations. **Recent Advances in Parallel Virtual Machine and Message Passing Interface: the ninth European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science**. 2472: 7

Gropp, W., S.H. Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir and M. Snir. 1998. **MPI: The Complete Reference Volumn 2 - The MPI-2 Extensions**. MIT Press.

Gropp, W., E. Lusk, N. Doss and A. Skjellum. 1996. A high-performance, portable implementation of the MPI message passing interface standard**. Parallel Computing** 22(6): 789-828.

Iannello, G. 1997. Efficient algorithms for the reduce-scatter operation in LogGP**. IEEE Transactions on Parallel and Distributed Systems**. 8(9): 970-982.

Kamal, H., B. Penoff and A. Wagner. 2005. SCTP versus TCP for MPI, pp. 30. In **The 2005 ACM/IEEE Conference on Supercomputing.** Seattle, WA.

Karp, R.M., A. Sahay, E.E. Santos and K.E. Schauser. 1993. Optimal broadcast and summation in the LogP model, pp. 142-153. In **The fifth Annual ACM Symposium on Parallel Algorithms and Architectures**. ACM Press New York, NY, USA.

Karonis, N.T., B. Supinski, I. Foster, W. Gropp and E. Lusk. 2002. A multilevel approach to topology-aware collective operations in computational Grids **. ArXiv Computer Science e-prints**: cs/0206038

Karonis, N.T., B. Toonen and I. Foster. 2003. MPICH-G2: A Grid-enabled implementation of the Message Passing Interface**. Journal of Parallel and Distributed Computing**. 63(5): 551-563.

Kielmann, T., H.E. Bal, S. Gorlatch, K. Verstoep and R.F.H. Hofman . 2001. Network performance-aware collective communication for clustered wide-area systems. **Parallel Computing**. 27(11): 1431-1456.

Kielmann T., R.F.H. Hofman, H.E. Bal, A. Plaat and R.A.F. Bhoedjang. 1999. MagPIe: MPI's collective communication operations for clustered wide area systems, pp. 131-140. In **The seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming**. ACM Press New York, NY, USA.

Le, T.T. and J. Rejeb . 2006. A detailed MPI communication model for distributed systems. **Future Generation Computer Systems**. 22 (3): 269-278.

Lee, C. and D. Talia. 2003. Grid programming models: Current tools, issues and directions, pp. 555-578. In **Grid Computing: Making the Global Infrastructure a Reality**. John Wiley and Sons Chichester, UK.

Levine, D. 1996. Users guide to the PGAPack parallel genetic algorithm library. Technical Report ANL-95/18, Argonne National Laboratory.

Matsuda, M., Y. Ishikawa, Y. Kaneo, M. Edamoto, F. Okazaki, H. Koie, R. Takano, T. Kudoh and Y. Kodama. 2005. Overview of the GridMPI Version 1.0, pp. In **Summer United Workshops on Parallel, Distributed and Cooperative Processing.**

Phatanapherom, S., P. Uthayopas. 2002. On the Building of a Job Scheduler System for Globus Grid Environment. In **APAN Conference 2002**. Shanghai, China.

Snir, M., S. Otto, S.H. Lederman, D. Walker and J. Dongarra. 1998. **MPI: The Complete Reference Volumn 1 - The MPI Core**. MIT Press

Squyres, J.M. and A. Lumsdaine. 2003. A component architecture for LAM/MPI. **Recent Advances in Parallel Virtual Machine and Message Passing Interface: the tenth European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science**. 2840: 379-387.

Stewart, R.R. and Q. Xie. 2001. **Stream Control Transmission Protocol (SCTP): A Reference Guide**. Addison-Wesley.

Tirumala, A., F. Qin, J. Dugan, and J. Ferguson. 2002. Iperf. http://dast.nlanr.net/ Projects/Iperf/.

Vadhiyar, S.S., G.E. Fagg and J. Dongarra . 2000. Automatically tuned collective communications , pp. 3. In **The 2000 ACM/IEEE Conference on Supercomputing.** IEEE Computer Society Washington, DC, USA

Varavidthaya, V. and P. Uthayopas. 2000. ThaiGrid: architecture and overview. **NECTEC Technical Journal**. 2(9):

Vorakosit, T. 2003. **Development of Robust and High Speed Message Passing Interface on Cluster Systems**. M.Eng. thesis, Kasetsart University

Vorakosit, T. and P. Uthayapos. 2003. Generating an Efficient Dynamics Multicast Tree under Grid Environment. **Recent Advances in Parallel Virtual Machine and Message Passing Interface: the tenth European PVM/MPI Users' Group Meeting, Lecture Notes in Computer Science**. 2840: 636-643

Vorakosit, T. and P. Uthayopas. 2004. Improving MPI multicast performance over Grid environment using intelligent message scheduling, pp 77-74. In **International Conference on Scientific and Engineering Computation.** Singapore.

Vorakosit, T. and P. Uthayopas, 2005. Building a Highly Scalable MPI Runtime Library on Grid using Hierarchical Virtual Cluster Approach. **In Proceedings of IASTED International Conference on Parallel and Distributed Computing Systems (PDCS2005**), Phoenix Arizona, USA

Vorakosit, T. and P. Uthayopas. 2005. Designing an Efficient Simulation Tool for MPI Runtime Algorithm on Grid, pp 92. In **The fifth International Conference on Information, Communications and Signal Processing**

Wu, J.J., S.h. Yeh and P. Liu. 2004. Efficient Multiple Multicast on Heterogeneous Network of Workstations**. The Journal of Supercomputing.** 29(1): 59-88

APPENDIX

APPENDIX

MPITH Reference Manual

MPI Namespace Reference

Namespace for MPI standard function.

1. Compounds

class Comm
Basic communicator class.

class Comm_Null
Null communicator.

class Datatype
Datatype identifier class.

class Group
Group of process class.

class Intracomm
Intra-communicator class.

class Op
Operation class for reduce operation.

class Status
Receiving status.

2. Type Definition

typedef u_int32_t MPI::Aint
Integer large enough to hold address of memory.

typedef u_int32_t MPI::TAG
TAG datatype.

typedef void(MPI::User_function)(void *invec, void *outvec, int *len, const Datatype & datatype)
Handler function for combine value.

Parameters:
invec first argument
outvec output array and second argument
len pointer to length of array

datatype reference to datatype of element in array

3. Function

MPI::Aint MPI::Address (const void * *location*)

Get the address of a location in memory.

Parameters:

location pointer to the location

Returns:

address of given location

void MPI::Finalize (void)

Terminates MPI execution environment.

bool MPI::Finalized (void)

Determine whether MPI_Finalize has been called.

Returns:

true, if MPI_Finalize has been already called. false, otherwise.

MPI::Aint MPI::Get_address (const void * *location*)

Get the address of a location in memory.

Parameters:

location pointer to the location

Returns:

address of given location

void MPI::Get_processor_name (char * *name*, int & *resultlen*)

Get processor name.

Parameters:

name pointer to contain name

resultlen reference to integer containing length of processor name

void MPI::Init (int & *argc*, char **& *argv*)

Initialize the MPI execution environment.

Parameters:

argc reference to the number of argument

argv reference to the argument array

bool MPI::Initialized (void)

Determine whether MPI_Init has been called.

Returns:

true, if MPI_Init has been already called. false, otherwise.

double MPI::Wtick (void)

Get the resolution of MPI::Wtime

Returns:
the resolution of Wtime

double MPI::Wtime (void)
Get an elapsed time on the calling processor.

Returns:
time in seconds since an arbitrary time in the past.

4. Variable Documentation

const int MPI::ANY_SOURCE
Predefined tag, matching any tag in receiving process.

const MPI::TAG MPI::ANY_TAG
Predefined tag, matching any tag in receiving process.

const MPI::Op MPI::BAND
bitwise and operator

const MPI::Datatype MPI::BOOL
Datatype representing boolean value.

const MPI::Op MPI::BOR
bitwise or operator

const MPI::Op MPI::BXOR
bitwise exclusive or operator

const MPI::Datatype MPI::BYTE
Datatype representing 8-bits unsigned integer.

const Datatype MPI::CHAR
Datatype representation of 8-bits signed character.

MPI::Comm_Null MPI::COMM_NULL
Null communicator.

MPI::Intracomm MPI::COMM_WORLD
Communicator containing all processes in this execution environment.

const MPI::Datatype MPI::DOUBLE
Datatype representing floating point double precision.

const int MPI::ERR_COMM
Invalid communicator.

const int MPI::ERR_COUNT
Invalid count argument.

const int MPI::ERR_RANK

Invalid rank argument.

const int MPI::ERR_TAG
Invalid tag argument.

const int MPI::ERR_TYPE
Invalid datatype argument.

const MPI::Datatype MPI::FLOAT
Datatype representing floating point single precision.

const int MPI::IDENT
Indentical, same order and member.

const Datatype MPI::INT
Datatype representing 32-bits signed integer.

const MPI::Op MPI::LAND
logical and operator

const MPI::Datatype MPI::LONG
Datatype representing 32-bits signed integer.

const MPI::Datatype MPI::LONG_DOUBLE
Datatype representing floating point long-double precision.

const MPI::Op MPI::LOR
logical or operator

const MPI::Op MPI::LXOR
logical exclusive or operator

const MPI::Op MPI::MAX
maximum operator

const MPI::Op MPI::MIN
minimum operator

const int MPI::MSG_TRUNCATE
In receiving process, indicate that receive buffer is not enough.

const MPI::Op MPI::NOP
no operation

const int MPI::PROC_NULL
Predefined rank, this rank may be used to send or receive from no-one.

const MPI::Op MPI::PROD
production operator

const MPI::Datatype MPI::SHORT
Datatype representing 16-bits signed integer.

const MPI::Datatype MPI::SIGNED_CHAR
Datatype representing 8-bits signed character.

const int MPI::SIMILAR
Similar, same only member.

const int MPI::SUCCESS
Successful return code.

const MPI::Op MPI::SUM
summation operator

const MPI::TAG MPI::TAG_ERROR
Invalid tag argument.

const int MPI::UNDEFINED
Used by many routine to indicated undefined or unknown integer value.

const int MPI::UNEQUAL
Different.

const MPI::Datatype MPI::UNSIGNED
Datatype representing 32-bits unsigned integer.

const MPI::Datatype MPI::UNSIGNED_CHAR
Datatype representing 8-bits unsigned character.

const MPI::Datatype MPI::UNSIGNED_LONG
Datatype representing 32-bits unsigned integer.

const MPI::Datatype MPI::UNSIGNED_SHORT
Datatype representing 16-bits unsigned integer.

MPITH Namespace Reference

Namespace for MPITH function.

1. Compounds

class Host
operation class.

class Process
MPITH process abstraction.

2. Variable Documentation

const int MPITH::PROCESS_FAIL
Predefined process state, running state.

const int MPITH::PROCESS_RUNNING
Predefined process state, running state.

MPI::Comm Class Reference

Basic communicator class.

1. Member Function Documentation

void MPI::Comm::Free (void)
Marks the communicator object for deallocatoin.

MPI::Group MPI::Comm::Get_group (void) const
Accesses the group associated with this communicator object.

Returns:
Group in this communicator

MPITH::Process MPI::Comm::Get_process (int *rank*) const
Get process object of specified rank.

Parameters:
rank rank of process
Returns:
Process object representation of given rank

int MPI::Comm::Get_rank (void) const
Determines the rank of the calling process in the communicator.

Returns:
rank of the calling process in group of this communicator object

int MPI::Comm::Get_size (void) const
Determine the size of the group associated with this communicator object.

Returns:
number of processes in the group of this communicator object

void MPI::Comm::Probe (int *source*, int *tag*) const
Blocking test for a message.

Parameters:
source rank of sources process
tag message tag

void MPI::Comm::Probe (int *source*, int *tag*, Status & *status*) const
Blocking test for a message.

Parameters:
source rank of sources process
tag message tag
status reference to status object for logging status of this probing

void MPI::Comm::Recv (void * *buf*, int *count*, const Datatype & *datatype*, int *source*, int *tag*) const
Performs a basic receive.

Parameters:
buf pointer to receive buffer
count number of elements in sent buffer
datatype of element in sent buffer
source rank of sender process
tag message tag

void MPI::Comm::Recv(void * *buf*, int *count*, const Datatype & *datatype*, int *source*, int *tag*, Status & *status*) const
Performs a basic receive.

Parameters:
buf pointer to receive buffer
count number of elements in sent buffer
datatype of element in sent buffer
source rank of sender process
tag message tag
status reference to status object for logging status of this receiving

void MPI::Comm::Send(const void * *buf*, int *count*, const Datatype & *datatype*, int *dest*, int *tag*) const
Performs a basic send.

Parameters:
buf pointer to sent buffer
count number of elements in sent buffer
datatype of element in sent buffer
dest rank of destination process or receiver
tag message tag

MPI::Request MPI::Comm::Isend(const void * *buf*, int *count*, const Datatype & *datatype*, int *dest*, int *tag*) const [virtual]
Begins a nonblocking send.

Parameters:
buf initial address of send buffer
count number of elements in send buffer

datatype datatype of each send buffer element
dest rank of destination
tag message tag
request communication request

bool MPI::Comm::Iprobe (int *source*, int *tag*) const [virtual]
Nonblocking test for a message.

Parameters:
source source rank, or MPI_ANY_SOURCE
tag value or MPI_ANY_TAG

bool MPI::Comm::Iprobe (int *source*, int *tag*, Status & *status*) const [virtual]
Nonblocking test for a message.

Parameters:
source source rank, or MPI_ANY_SOURCE
tag value or MPI_ANY_TAG
status status object

MPI::Request MPI::Comm::Irecv (void * *buf*, int *count*, const Datatype & *datatype*, int *dest*, int *tag*) const [virtual]
Begins a nonblocking receive.

Parameters:
buf initial address of receive buffer
count number of elements in receive buffer
datatype datatype of each receive buffer element
source rank of source
tag message tag
request communication request

MPI::Request MPI::Comm::Isend (const void * *buf*, int *count*, const Datatype & *datatype*, int *dest*, int *tag*) const [virtual]
Begins a nonblocking send.

Parameters:
buf initial address of send buffer
count number of elements in send buffer
datatype datatype of each send buffer element
dest rank of destination
tag message tag
request communication request

MPI::Request MPI::Comm::Issend (const void * *buf*, int *count*, const Datatype & *datatype*, int *dest*, int *tag*) const [virtual]
Start a nonblocking synchronous send.

Parameters:
buf initial address of send buffer

count number of elements in send buffer
datatype datatype of each send buffer element
dest rank of destination
tag message tag
request communication request

void MPI::Comm::Sendrecv (const void * *sendbuf*, int *sendcount*, const Datatype & *sendtype*, int *dest*, int *sendtag*, void * *recvbuf*, int *recvcount*, const Datatype & *recvtype*, int *source*, int *recvtag*) const [virtual]

Sends and receives a message.

Parameters:

sendbuf initial address of send buffer
sendcount number of elements in send buffer
sendtype type of elements in send buffer
dest rank of destination
sendtag send tag
recvbuf initial address of receive buffer
recvcount number of elements in receive buffer
recvtype type of elements in receive buffer
source rank of source
recvtag receive tag

void MPI::Comm::Sendrecv (const void * *sendbuf*, int *sendcount*, const Datatype & *sendtype*, int *dest*, int *sendtag*, void * *recvbuf*, int *recvcount*, const Datatype & *recvtype*, int *source*, int *recvtag*, Status & *status*) const [virtual]

Sends and receives a message.

Parameters:

sendbuf initial address of send buffer
sendcount number of elements in send buffer
sendtype type of elements in send buffer
dest rank of destination
sendtag send tag
recvbuf initial address of receive buffer
recvcount number of elements in receive buffer
recvtype type of elements in receive buffer
source rank of source
recvtag receive tag
status status object. This refers to the receive operation

void MPI::Comm::Sendrecv_replace (void * *buf*, int *count*, const Datatype & *datatype*, int *dest*, int *sendtag*, int *source*, int *recvtag*) const [virtual]

Sends and receives using a single buffer.

Parameters:

buf initial address of send and receive buffer
count number of elements in send and receive buffer
datatype type of elements in send and receive buffer

dest rank of destination
sendtag send message tag
source rank of source
recvtag receive message tag

void MPI::Comm::Sendrecv_replace (void * *buf*, int *count*, const Datatype & *datatype*, int *dest*, int *sendtag*, int *source*, int *recvtag*, Status & *status*) const [virtual]
Sends and receives using a single buffer.

Parameters:
buf initial address of send and receive buffer
count number of elements in send and receive buffer
datatype type of elements in send and receive buffer
dest rank of destination
sendtag send message tag
source rank of source
recvtag receive message tag
status status object

void MPI::Comm::Ssend (const void * *buf*, int *count*, const Datatype & *datatype*, int *dest*, int *tag*) const [virtual]
Performs basic synchronus send.

Parameters:
buf initial address of send buffer
count number of elements in send buffer
datatype datatype of each send buffer element
dest rank of destination
tag message tag

MPI::Datatype Class Reference

MPI::Datatype Datatype identifier class.

1. Member Function Documentation

void MPI::Datatype::Commit (void) const
Commits the datatype. All user-specified datatype must be commited before use.

MPI::Datatype MPI::Datatype::Create_struct (int *count*, const int *blklen*[], const Aint *disp*[], const Datatype *types*[]) [static]
Create a struct datatype.

Parameters:
count number of block -- also the number of entires of blklen and disp
blklen array containing number of element in each block
disp array containing byte displacement of each block

types array containing datatype of element in each block

Returns:

new structure datatype

void MPI::Datatype::Free (void)

Free this datatype object.

MPI::Aint MPI::Datatype::Get_extent (void) const

Get the extent of a datatype. The extend of datatype is computed by Get_ub() - Get_lb();

Returns:

extent of a datatype.

MPI::Aint MPI::Datatype::Get_lb (void) const

Get the lower-bound of this datatype.

Returns:

lower-bound of this datatype.

int MPI::Datatype::Get_size (void) const

Get the number of bytes occupied by entries in the datatype.

Returns:

size of this datatype.

MPI::Aint MPI::Datatype::Get_ub (void) const

Get the upper-bound of this datatype.

Returns:

upper-bound of this datatype

MPI::Group Class Reference

MPI::Group Group of process class.

1. Member Function Documentation

int MPI::Group::Compare (const Group & *group1*, const Group & *group2*) [static]

Comparing two groups.

Parameters:

group1 the first group to be compared

group2 the second group to be compared

Returns:

MPI_INDENT, MPI_SIMILAR, or MPI_UNEQUAL if the order and members of two groups are same, only members are same and otherwise respectively.

MPI::Group MPI::Group::Excl (int *n*, const int *ranks*[]) const

Produces a group by reordering this group and taking only ranks not listed in ranks array.

Parameters:

n number of elements in ranks

ranks array containing rank of process to not appear in new group

Returns:

new group containing only processes in not ranks

void MPI::Group::Free (void)

Frees this group.

int MPI::Group::Get_rank (void) const

Get rank of calling process in this group.

Returns:

rank of calling process in this group

int MPI::Group::Get_size (void) const

Get size of this group.

Returns:

number of processes in this group

MPI::Group MPI::Group::Incl (int *n*, const int *ranks*[]) const

Produces a group by reordering this group and taking only ranks listed in ranks array.

Parameters:

n number of elements in ranks

ranks array containing rank of process to appear in new group

Returns:

new group containing only processes in ranks

MPITH::Host Class Reference

MPITH::Host Host operation class.

1. Member Function Documentation

MPITH::Host MPITH::Host::Get_host (const MPI::Comm & *comm*, int *rank*) [static]

Get host object from communicator and rank.

Parameters:

comm communicator containing target host

rank ranks of process to be get

Returns:

host object

MPITH::Host MPITH::Host::Get_host (void * *addr*, int *addr_len*) [static]

Get host object from specify host address.

Parameters:
addr pointer to host address
length of host address
Returns:
host object

int MPITH::Host::Probe (void)
Probe whether this host is online.

Returns:
0 if host is online, less than 0 otherwise.

MPI::Intracomm Class Reference

MPI::Intracomm Intracommunicator class. Inheritance diagram for MPI::Intracomm:

1. Member Function Documentation

void MPI::Intracomm::Barrier (void) const
Block until all processes in this communicator have reached this routine.

void MPI::Intracomm::Bcast (void * *buf*, int *count*, const Datatype & *datatype*, int *root*) const
Broadcasts a message from root process to all process in this communicator.

Parameters:
buf pointer to buffer containing message to be broadcasted
count number of element in buf
datatype datatype in buf
root rank of the root process

int MPI::Intracomm::bcastMpidTable (int *root*)
Broadcast mpid table to every node in world at initialize stage.

MPI::Intracomm MPI::Intracomm::Dup (void) const
Duplicate this intracommunicator.

Returns:
new communicator object

void MPI::Intracomm::Gather (const void * *sendbuf*, int *sendcnt*, const Datatype & *stype*, void * *recvbuf*, int *recvcnt*, const Datatype & *rtype*, int *root*) const
Gathers together values from all processes in this communicator.

Parameters:
sendbuf pointer to send buffer
sendcnt number of elements in sendbuf
stype datatype of elements in sendbuf
recvbuf pointer to receive buffer, significant only at root
recvcnt number of elements in recvbuf, significant only at root

rtype datatype of elements in recvbuf
root rank of root process

void MPI::Intracomm::Reduce (const void * *sendbuf*, void * *recvbuf*, int *count*, const Datatype & *datatype*, const Op & *op*, int *root*) const

Reduces values on all processes in this group to a single value in root process.

Parameters:

sendbuf pointer to array containing data to be reduced
recvbuf pointer to receive buffer, significant only at root
count number of elements in sendbuf
datatype datatype of elements in sendbuf
op operation to perform on sendbuf
root rank of root process

void MPI::Intracomm::Scatter (const void * *sendbuf*, int *sendcount*, const Datatype & *sendtype*, void * *recvbuf*, const int *recvcount*, const Datatype & *recvtype*, int *root*) const

Sends data from one task to all other tasks in a group.

Parameters:

sendbuf pointer to sending buffer
sendcount number of element for EACH process
sendtype datatype of sent items
recvbuf pointer to receiving buffer
recvtype datatype of received items
root root of this operation

void MPI::Comm::Abort (int *errorcode*) [virtual, inherited]

Terminates MPI execution environment.

Parameters:

errorcode error code to be returned to invoking environment

void MPI::Intracomm::Allgather (const void * *sendbuf*, int *sendcount*, const Datatype & *sendtype*, void * *recvbuf*, int *recvcount*, const Datatype & *recvtype*) const [virtual]

Gathers data from all tasks and distribute it to all.

Parameters:

sendbuf starting address of send buffer
sendcount number of elements in send buffer
sendtype data type of send buffer elements
recvcount number of elements received from any process
recvtype data type of receive buffer elements
comm communicator
recvbuf address of receive buffer

void MPI::Intracomm::Alltoall (const void * *sendbuf*, int *sendcount*, const Datatype & *sendtype*, void * *recvbuf*, int *recvcount*, const Datatype & *recvtype*) const [virtual]

Sends data from all to all process.

Parameters:

sendbuf starting address of send buffer (choice)
sendcount number of elements to send to each process (integer)
sendtype data type of send buffer elements (handle)
recvcount number of elements received from any process (integer)
recvtype data type of receive buffer elements (handle)
comm communicator (handle)
recvbuf address of receive buffer (choice)

MPI::Op Class Reference

MPI::Op Operation class for reduce operation.

1. Member Function Documentation

void MPI::Op::compute (void * *data1*, void * *data2*, int *count*, const MPI::Datatype * *dt*) const

Compute its operation in the form data1 = data1 op data2.

Parameters:

data1 pointer to first data buffer
data2 pointer to other data buffer
count number of datatype to be compute
d1 pointer to datatype object pointed by data1

void MPI::Op::Free (void)

Free this operator.

void MPI::Op::Init (MPI::User_function * *func*, bool *commute*)

Initialized this operation object.

Parameters:

func pointer to user function
commute flag whether this operation has property of commutative

MPITH::Process Class Reference

MPITH::Process MPITH process abstraction.

1. Member Function Documentation

MPITH::mpipid_t MPITH::Process::Create (void) [static]

Create new process.

Returns:

MPITH process identifier.

MPITH::Host MPITH::Process::Get_host (void) const
Get the host that this process is running.

Returns:
host object

int MPITH::Process::Probe (void) const
Probe for a status of process.

Returns:
process status

MPI::Status Class Reference

MPI::Status Receiving status.

1. Member Function Documentation

int MPI::Status::Get_count (const Datatype & *datatype*) const
Gets the number of "top level" elements.

Parameters:
datatype datatype to be a reference
Returns:
number of element

int MPI::Status::Get_elements (const Datatype & *datatype*) const
Get the number of basic elements.

Parameters:
datatype datatype used by receive operation
Returns:
number of basic elements

int MPI::Status::Get_error (void) const
Get error condition.

Returns:
error condition

int MPI::Status::Get_source (void) const
Get sender.

Returns:
rank of sender process

int MPI::Status::Get_tag (void) const
Get tag.

Returns:
tag of received message

CURRICULUM VITAE

NAME:	Mr. Theewara Vorakosit		
BIRTH DATE:	May 1, 1979		
BIRTH PLACE	Bangkok, Thailand		
EDUCATION:	YEAR	INSTITUTION	DEGREE/DIPLOMA
	2001	Kasetsart University	B. Eng. Hons. (Computer Engineering)
	2003	Kasetsart University	M. Eng. (Computer Engineering)
POSITION/TITLE:	Assistant Researcher		
WORK PLACE	High Performance Computing and Networking Center, Faculty of Engineering, Kasetsart University.		