

# Automatic Generation and Tuning of MPI Collective Communication Routines \*

Ahmad Faraj      Xin Yuan  
Department of Computer Science, Florida State University  
Tallahassee, FL 32306  
{faraj, xyuan}@cs.fsu.edu

## ABSTRACT

In order for collective communication routines to achieve high performance on different platforms, they must be able to adapt to the system architecture and use different algorithms for different situations. Current Message Passing Interface (MPI) implementations, such as MPICH and LAM/MPI, are not fully adaptable to the system architecture and are not able to achieve high performance on many platforms. In this paper, we present a system that produces efficient MPI collective communication routines. By automatically generating topology specific routines and using an empirical approach to select the best implementations, our system adapts to a given platform and constructs routines that are customized for the platform. The experimental results show that the tuned routines consistently achieve high performance on clusters with different network topologies.

## Categories and Subject Descriptors

D.1.3 [Programming Techniques]: Concurrent Programming—*Distributed Programming*

## General Terms

Performance

## Keywords

MPI, Cluster of Workstations, Tuning System, Empirical

## 1. INTRODUCTION

The Message Passing Interface (MPI) [15] provides a simple communication API and eases the task of developing portable parallel applications. Its standardization has resulted in the development of a large number of MPI based parallel applications. For these applications to achieve high

\*This work was partially supported by NSF grants ANI-0106706, CCR-0208892, and CCF-0342540.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICS '05, June 20–22, Boston, MA, USA.

Copyright © 2005, ACM 1-59593-167-8/06/2005 ...\$5.00.

performance, it is crucial that the MPI library realizes the communications efficiently.

In this paper, we consider MPI collective communication routines, where multiple nodes participate in the communication operation. MPI developers have long recognized the need for the communication routines to be adaptable to the system architecture and/or the application communication patterns in order to achieve high performance on a system. Various forms of software adaptability are supported in most MPI implementations. For example, MPICH-1.2.6 [21] utilizes different algorithms based on message size when realizing all-to-all operation. However, the software adaptability supported in the current MPI libraries, including MPICH [16, 21] and LAM/MPI [9], is insufficient and these libraries are not able to achieve high performance on many platforms.

There are inherent limitations in the current implementations of MPI collective communication routines. First, since the library routines are implemented before the topology information is known, it is impossible for the library to utilize topology specific algorithms. Using topology unaware algorithms can generally perform reasonably well when the message size is small since the network can handle such cases without significantly degrading the performance. However, when the message size is large, the network contention problem can significantly affect the communication performance. This is particularly true when nodes are not connected by a single crossbar switch. Second, for any communication algorithm, there are many system parameters that can affect the performance of the algorithm. These parameters, which include operating system context switching overheads, the ratio between the network and the processor speeds, the switch design, the switch buffer capacity, and the network topology, are difficult to model. The library developer cannot make the right choices for different platforms.

In this paper, we present a system that overcomes these limitations. The system is based on two main techniques. First, topology specific communication routines are automatically generated by a routine generator that takes the topology information as input. The routines are added to an algorithm repository maintained by the system, which also includes an extensive number of topology unaware algorithms for each supported routine. Second, an empirical approach is used to select the best implementation among the different topology specific and topology unaware algorithms. These two techniques enable our system to adapt to different architectures and construct efficient collective communication routines that are customized to the architectures.

The tuning system is developed for Ethernet switched clusters. It currently tunes five MPI collective communication routines: *MPI\_Alltoall*, *MPI\_Alltoallv*, *MPI\_Allgather*, *MPI\_Allgatherv*, and *MPI\_Allreduce*. The routines produced by the system run on LAM/MPI [9]. The experimental results show that the tuned routines are very robust and yield good performance for clusters with different network topologies. The tuned routines sometimes out-perform the routines in LAM/MPI and MPICH [21] to a very large degree.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes the automatic generation and tuning system. Section 4 reports the performance results, and Section 5 concludes the paper.

## 2. RELATED WORK

The success of the MPI standard can be attributed to the wide availability of two MPI implementations: MPICH[7, 16, 21] and LAM/MPI [9]. Many researchers have been trying to optimize the MPI library [10, 12, 13, 17, 19, 20, 21]. In [13], optimizations are proposed for collective communications over Wide-Area Networks by considering the network details. In [17], a compiler based optimization approach is developed to reduce the software overheads in the library. In [10], MPI point-to-point communication routines are optimized using a more efficient primitive (Fast Message). Optimizations for a thread-based MPI implementation are proposed in [20]. Optimizations for clusters of SMPs are presented in [19]. A combined compiler and library approach was proposed in [12]. Our system differs in that it constructs high performance collective communication routines by automatically generating topology specific routines and by empirically selecting the best algorithms. The algorithm repository maintained in our system includes many algorithms developed by various research groups [2, 4, 11, 12, 14, 21]. This paper, however, focuses on the automatic generation and the automatic tuning of the algorithms, not on the individual algorithm development.

The empirical tuning technique used in our system is a variation of the Automated Empirical Optimization of Software (AEOS) technique [23]. The idea of AEOS is to optimize software automatically using an empirical approach that includes timers, search heuristics, and various methods of software adaptability. This technique has been applied successfully to various computational library routines [1, 6, 23]. The research that is closely related to our work is presented in [22], where the AEOS technique was applied to optimize collective communications. Our work differs from [22] in a number of ways. First, our system considers algorithms that are specific to the physical topology while algorithms in [22] use logical topologies and are unaware of the physical topology. Second, the system in [22] tries to tune and produce common routines for systems with different numbers of nodes. Our system is less ambitious in that we tune routines for a specific physical topology. By focusing on a specific physical topology, we are able to construct high efficiency routines. Third, [22] mainly focused on one-to-all and one-to-many communications and studied various message pipelining methods to achieve the best performance. This paper considers all-to-all and many-to-many communications where pipelining is not a major factor that affects the communication performance. For the types of communications investigated in this paper, selecting the right algorithm for a given system and communication pattern is crucial for

achieving high performance.

## 3. AUTOMATIC GENERATION AND TUNING SYSTEM

The automatic generation and tuning system is designed to construct efficient collective communication routines that are customized for a particular platform and/or application. For each communication routine supported by the system, an extensive set of topology unaware and topology specific algorithms is maintained. The system utilizes an empirical approach to determine the best algorithms among all algorithms in the set for the communication operation under different conditions. The combination of the automatic generation of topology specific routines and the empirical approach enables the system to fully adapt to a platform and construct efficient customized routines.

Currently, the system supports five routines: *MPI\_Alltoall*, *MPI\_Alltoallv*, *MPI\_Allgather*, *MPI\_Allgatherv*, and *MPI\_Allreduce*. The tuned routines use point-to-point primitives in LAM/MPI. The system can be extended to produce routines that run on any communication library that provides MPI-like point-to-point communication primitives. Currently, the system only provides automatic routine generator for homogeneous Ethernet switched clusters. Hence, the full system only works on Ethernet switched clusters. The system can tune topology unaware algorithms on any platform that supports LAM/MPI and allows users to supply their own topology specific routine generators, potentially for other types of networks.

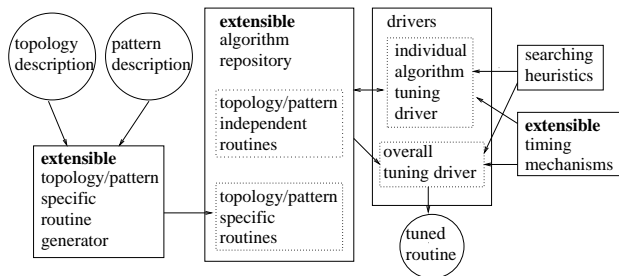


Figure 1: System overview

As shown in Figure 1, there are five major components in the system: the extensible topology/pattern specific routine generator, the extensible algorithm repository, the search heuristics, the extensible timing mechanisms, and the drivers. The extensible topology/pattern specific routine generator takes topology description and sometimes pattern description and generates topology/pattern specific routines. For *MPI\_Alltoall*, *MPI\_Allgather*, and *MPI\_Allreduce*, only the topology description is needed. The pattern description is also needed for *MPI\_Alltoallv* and *MPI\_Allgatherv*. This module is extensible in that users can provide their own routine generators with their own topology descriptors and pattern descriptors to replace the system built-in generators. For each supported MPI routine, the algorithm repository contains an extensive set of algorithms to be used in the tuning process. These algorithms include system built-in topology/pattern unaware algorithms, topology/pattern specific algorithms generated by the routine generator module, and the results from the individual algorithm tuning drivers. Each routine in the repository may have zero, one,

or more algorithm parameters. The repository is extensible in that it allows users to add their own implementations. The search heuristics determine the order in the search of the parameter space for deciding the best values for algorithm parameters. The extensible timing mechanisms determine how the timing results are measured. This module is extensible in that users can supply their own timing mechanisms. The driver module contains individual algorithm tuning drivers and overall tuning driver. The individual algorithms tuning drivers tune algorithms with parameters, produce routines with no parameters (parameters are set to optimal values), and store the tuned routines back in the algorithm repository. The overall tuning driver considers all algorithms with no parameters and produces the final tuned routine. Next, we will describe each module in more details.

### 3.1 Algorithm repository

We will first describe the cost model that we use to give a rough estimate of the communication performance for the algorithms. It must be noted that some parameters in the cost model that can contribute significantly to the overall communication costs, such as sequentialization costs and network contention costs described below, are very difficult to quantify. In practice, they cannot be measured accurately since they are non-deterministic in nature. As a result, this cost model can only be used to guide the selection of algorithms in the tuning system repository, but cannot be used to predict accurately which algorithm will be most effective for a given system setting. Our tuning system uses an empirical approach to select the most effective implementations, and can even operate without the cost model. After discussing the cost model, we will then introduce the algorithms maintained in the repository, including both topology unaware algorithms and topology specific algorithms.

#### Cost Model

The model reflects the following costs:

- *Per pair communication time.* The time taken to send a message of size  $n$  bytes between any two nodes can be modeled as  $\alpha + n\beta$ , where  $\alpha$  is the startup overhead, and  $\beta$  is the per byte transmission time.
- *Sequentialization overhead.* Some algorithms partition the all-to-all type of communication into a number of phases. A communication in a phase can only start after the completion of some communications in the previous phases. This sequentialization overhead may limit the parallelism in the communication operation. We use  $\theta$  to denote the sequentialization overhead between 2 phases. For a communication with  $m$  phases, the sequentialization overhead is  $(m - 1)\theta$ .
- *Synchronization overhead.* We use two types of synchronizations in our algorithms: *light-weight*  $\delta_l$  and *heavy-weight*  $\delta_h$ . A light-weight barrier ensures that a communication happens before another while heavy-weight barrier uses a system wide synchronization by calling *MPIBarrier*. In most cases,  $\delta_h$  is larger than  $\delta_l$ , which is larger than  $\theta$ .
- *Contention overhead.* Contention can happen in three cases: *node contention*  $\gamma_n$  when multiple nodes send to the same receiver, *link contention*  $\gamma_l$  when multiple communications use the same network links, and *switch contention*  $\gamma_s$  when the amount of data passing a switch is more than the switch capacity. We will use  $\gamma = \gamma_n + \gamma_l + \gamma_s$  to denote the sum of all contention costs.

Using this model, the time to complete a collective com-

munication is expressed in the above five terms ( $\alpha$ ,  $\beta$ ,  $\theta$ ,  $\delta$ , and  $\gamma$ ). The startup time and sequentialization overhead terms are important for algorithms for small messages while the bandwidth, synchronization costs, and contention overhead terms are important for algorithms for large messages.

In the rest of the paper, we will assume that  $p$  is the number of processes and  $n$  is the message size (passed as a parameter to routines *MPIAlltoall*, *MPIAllgather*, and *MPIAllreduce*, that is,  $n = \text{sendcount} * \text{size\_of\_element}$ ). Each node in the system can send and receive a message simultaneously, which is typical in Ethernet switched clusters.

#### Algorithms for MPIAlltoall

In the following, we assume by default that an algorithm does not have parameters, unless specified otherwise.

**Simple algorithm.** This algorithm basically posts all receives and all sends, starts the communications, and waits for all communications to finish. Let  $i \rightarrow j$  denote the communication from node  $i$  to node  $j$ . The order of communications for node  $i$  is  $i \rightarrow 0, i \rightarrow 1, \dots, i \rightarrow p - 1$ . The estimated time for this algorithm is  $(p - 1)(\alpha + n\beta) + \gamma$ .

**Spreading Simple algorithm.** This is similar to the simple algorithm except that the order of communications for node  $i$  is  $i \rightarrow i + 1, i \rightarrow i + 2, \dots, i \rightarrow (i + p - 1) \bmod p$ . This communication order may potentially reduce node contention. The estimated time is the same as that for simple algorithm except that the  $\gamma$  term might be smaller.

**2D mesh algorithm.** This algorithm organizes the nodes as a logical  $x \times y$  mesh and tries to find the factoring such that  $x$  and  $y$  are close to  $\sqrt{p}$ . The all-to-all operation is carried out first in the  $x$  dimension and then in the  $y$  dimension. For all data to reach all nodes, the all-to-all operation is actually an all-gather operation that collects all data from each node to all nodes in each dimension. Thus, assume  $x = y = \sqrt{p}$ , the message size for the all-gather operation in the  $x$  dimension is  $pn$  and the message size for the all-gather operation in the  $y$  dimension is  $p\sqrt{pn}$ . The estimated time for this algorithm is  $(\sqrt{p} - 1)(\alpha + pn\beta) + (\sqrt{p} - 1)(\alpha + p\sqrt{pn}\beta) + \theta + \gamma = 2(\sqrt{p} - 1)\alpha + (p - 1)pn\beta + \theta + \gamma$ . Compared to the simple algorithms, the 2D mesh algorithm sends less messages, but more data. There is a  $\theta$  term in the estimated time since communications are carried out in two phases.

**3D mesh algorithm.** This algorithm organizes the nodes as a logical  $x \times y \times z$  mesh. Assume  $x = y = z = \sqrt[3]{p}$ . The estimated time is  $3(\sqrt[3]{p} - 1)\alpha + (p - 1)pn\beta + 2\theta + \gamma$ . Compared to 2D mesh algorithm, this algorithm sends lesser messages, but consists of three phases, which introduce a  $2\theta$  sequentialization overhead.

**Recursive doubling (rdb) algorithm.** When the number of processes is a power of two, the recursive doubling algorithm is the the extension of the 2D mesh and 3D mesh algorithms to the extreme: a  $lg(p)$ -dimensional mesh with 2 nodes in each dimension. This algorithm first performs an all-gather operation to collect all data from all nodes to each node. Each node then copies the right portion of the data to its receiving buffer. Details about recursive doubling can be found in [21]. When the number of nodes is a power of two, the estimated time is  $lg(p)\alpha + (p - 1)pn\beta + (lg(p) - 1)\theta + \gamma$ . When the number of processes is not a power of two, the cost almost doubles [21]. Compared to the 3D mesh algorithm, this algorithm has a smaller startup time, but larger sequentialization overhead.

**Bruck algorithm.** This is another  $lg(p)$ -step algorithm that

sends less extra data in comparison to the recursive doubling algorithm. Details can be found in [2, 21]. When the number of processes is a power of two, the estimated time is  $lg(p)\alpha + \frac{n^2}{2}lg(p)\beta + (lg(p) - 1)\theta + \gamma$ . This algorithm also works with slightly larger overheads when the number of processes is not a power of two.

The above algorithms are designed for communication of small messages. Thus, the bandwidth and the contention terms in the estimated time are insignificant. To achieve good performance, the best trade-off must be found between the startup overhead and the sequentialization overhead. Next, we will discuss algorithms designed for large messages.

**Ring algorithm.** This algorithm partitions the all-to-all communication into  $p - 1$  steps (phases). In step  $i$ , node  $j$  sends a messages to node  $(j + i) \bmod p$  and receives a message from node  $(j - i) \bmod p$ . Thus, this algorithm does not incur node contention if all phases are executed in a lock-step fashion. Since different nodes may finish a phase and start a new phase at different times, the ring algorithm only reduces the node contention (not eliminates it). The estimated time is  $(p - 1)(\alpha + n\beta) + (p - 2)\theta + \gamma_n + \gamma_s + \gamma_l$ .

**Ring with light barrier algorithm.** This algorithm adds light-weight barriers between the communications in different phases that can potentially cause node contention and eliminates such contention. The estimated time is  $(p - 1)(\alpha + n\beta) + (p - 2)\delta_l + \gamma_s + \gamma_l$ . Compared to the ring algorithm, this algorithm incurs overheads for the light-weight barriers while reducing the contention overheads.

**Ring with MPI barrier algorithm.** The previous algorithm allows phases to proceed in an asynchronous manner which may cause excessive data injected into the network. The ring with MPI barrier algorithm adds an MPI barrier between two phases and makes the phases execute in a lock-step fashion, resulting in a less likely switch contention. The estimated time is  $(p - 1)(\alpha + n\beta) + (p - 2)\delta_h + \gamma_l$ . Compared to the ring with light barrier algorithm, this algorithm incurs heavy-weight synchronization overheads while reducing the switch contention overheads.

**Ring with  $N$  MPI barriers algorithm.** Adding a barrier between every two phases may be an over-kill and may result in the network being under-utilized since most networks and processors can effectively handle a certain degree of contention. The ring with  $N$  MPI barriers algorithm adds a total of  $1 \leq N\_mpi\_barrier \leq p - 2$  barriers in the whole communication (a barrier is added every  $\frac{p-1}{N\_mpi\_barrier+1}$  phases). This allows the contention overheads and the synchronization overheads to be compromised. The estimated time for this algorithm is  $(p - 1)(\alpha + n\beta) + N\delta_h + \gamma_n + \gamma_s + \gamma_l$ . This algorithm has one parameter, the number of barriers ( $N\_mpi\_barrier$ ). The potential value is in the range of 1 to  $p - 2$ .

**Pair algorithm.** The algorithm only works when the number of processes is a power of two. This algorithm partitions the all-to-all communication into  $p - 1$  steps. In step  $i$ , node  $j$  sends and receives a message to and from node  $j \oplus i$  (exclusive or). The estimated time is the same as that for the ring algorithm. However, in the pair algorithm, each node interacts with one other node in each phase compared to two in the ring algorithm. The reduction of the coordination among the nodes may improve the overall communication efficiency. Similar to the ring family algorithms, we have **pair with light barrier**, **pair with MPI barrier**, and

**pair with  $N$  MPI barriers** algorithms.

The ring family and the pair family algorithms try to remove node contention and indirectly reduce other contention overheads by adding synchronizations to slow down communications. These algorithms are topology unaware and may not be sufficient to eliminate link contention since communications in one phase may share the same link in the network. The topology specific algorithm removes link contention by considering the network topology.

**Topology specific algorithm.** We use a message scheduling algorithm that we developed in [4]. This algorithm finds the optimal message scheduling by partitioning the all-to-all communication into phases such that communications within each phase do not have contention, and a minimum number of phases are used to complete the communication. The algorithm estimated time depends on the topology.

### Algorithms for MPI\_Allgather

Since the *MPI\_Allgather* communication pattern is a special all-to-all communication pattern, most of the all-to-all algorithms can be applied to perform an all-gather operation. Our system includes the following all-gather algorithms that work exactly like their all-to-all counterparts (same estimated time), **simple**, **spreading simple**, **ring**, **ring with light barrier**, **ring with MPI barrier**, **ring with  $N$  MPI barriers**, **pair**, **pair with light barrier**, **pair with MPI barrier**, and **pair with  $N$  MPI barriers**. The following all-gather algorithms have different estimated times from their all-to-all counterparts: **2D mesh** with an estimated time of  $2(\sqrt{p} - 1)\alpha + (p - 1)n\beta + \theta + \gamma$ , **3D mesh** with an estimated time of  $3(\sqrt[3]{p} - 1)\alpha + (p - 1)n\beta + 2\theta + \gamma$ , and **Recursive doubling (rdb)**. When the number of processes is a power of two, the estimated time of rdb is  $lg(p)\alpha + (p - 1)n\beta + (lg(p) - 1)\theta + \gamma$ . The repository also includes the following algorithms:

**Bruck algorithm.** The Bruck all-gather algorithm is different from the Bruck all-to-all algorithm. Details can be found in [2, 21]. When the number of processes is a power of two, the estimated time is similar to the recursive doubling algorithm. The time is better than that of recursive doubling when the number of processes is not a power of two.

**Gather-Bcast algorithm.** This algorithm first gathers all data to one node and then broadcasts the data to all nodes. Assume that the gather and broadcast operations use the binary tree algorithm, the estimated time is  $lg(p)(\alpha + n\beta) + (lg(p) - 1)\theta + \gamma$  for gather and  $lg(p)(\alpha + pn\beta) + (lg(p) - 1)\theta + \gamma$  for broadcast.

**Topology specific algorithm.** We use an all-gather algorithm that we developed in [3]. The algorithm numbers the switches based on the order the switches are visited in Depth First Search (DFS). Let switch  $s_i$  be the  $i$ -th switch visited in DFS and let the machines attached to switch  $s_i$  be  $n_{i,0}, n_{i,1}, \dots, n_{i,m_i}$ . Assume that the system has switches  $s_0, s_1, \dots, s_k$ . The algorithm uses the following logical ring (LR) communication pattern to realize the all-gather operation:  $n_{0,0} \rightarrow n_{0,1} \rightarrow \dots \rightarrow n_{0,m_0} \rightarrow n_{1,0} \rightarrow n_{1,1} \rightarrow \dots \rightarrow n_{1,m_1} \rightarrow \dots \rightarrow n_{k,0} \rightarrow n_{k,1} \rightarrow \dots \rightarrow n_{k,m_k} \rightarrow n_{0,0}$ . As proven in [3], this logical ring pattern is contention free. To complete an all-gather operation, the algorithm repeats the logical ring communication pattern  $p - 1$  times. In the first iteration, each node sends its own data to the next adjacent node in the logical ring. In the following iterations, each node forwards what it received in the previous iteration to

its adjacent node. Details about this algorithm can be found in [3]. The estimated time is  $(p-1)(\alpha+n\beta)+(p-2)\theta+\gamma_s$ . Note that MPICH [16] uses a topology unaware logical ring algorithm that operates in the same way as our algorithm. However, without considering the network topology, the ring pattern in the MPICH algorithm may result in severe network contention, which degrades the performance.

### Algorithms for MPI\_Allreduce

Our implementations for *MPI\_Allreduce* assume the reduction operation is commutative.

**Reduce-Bcast** algorithm. The algorithm first performs a reduction to a node and then broadcasts the results to all nodes. Assuming that the binary tree algorithm is used for both operations, the time for both operations is  $lg(p)(\alpha+n\beta)+(lg(p)-1)\theta+\gamma$ .

**All-gather based** algorithm. The algorithm first gathers all data to all nodes. Then, each node performs the reduction locally. This algorithm uses the tuned *MPI\_Allgather* routine, which can be topology specific.

**Recursive doubling (rdb)** algorithm. This algorithm is similar to the all-gather based algorithm except that the reduction operation is performed while the data are being distributed. Since we ignore the computation costs, the estimated time is the same as recursive doubling for all-gather.

**MPICH Rabenseifner** algorithm. This algorithm completes in two phases: a reduce-scatter followed by an all-gather. The reduce-scatter is realized by recursive halving, which has a similar estimated time as recursive doubling. The all-gather is realized by recursive doubling. The time for this algorithm is roughly 2 times that of rdb algorithm for all-gather with a message size of  $\frac{n}{p}$ . More details of the algorithm can be found in [14].

**Rabenseifner** algorithm variation 1. This is a Rabenseifner algorithm with the all-gather operation performed using the tuned all-gather routine. This algorithm may be topology specific since the tuned all-gather routine may be topology specific.

**Rabenseifner** algorithm variation 2. In this variation, the reduce-scatter operation is realized by the tuned all-to-all routine with a message size of  $\frac{n}{p}$  and the all-gather operation is realized by the tuned all-gather routine with a message size of  $\frac{n}{p}$ .

### Algorithms for MPI\_Alltoallv

Most of the topology unaware all-to-all algorithms can be used to realize the *MPI\_Alltoallv* operation. Our system contains the all-to-allv version of the following all-to-all algorithms: **simple**, **spreading simple**, and the **ring** and **pair** families algorithms.

**Topology specific** algorithms. There are two topology specific *MPI\_Alltoallv* algorithms: **greedy** algorithm and **all-to-all based** algorithm. These two algorithms are extensions of the algorithms with the same names in the CCMPI package we developed [12]. Since *MPI\_Alltoallv* supports many-to-many communication with different message sizes, there are three issues in realizing this communication: balancing the load, reducing network contention, and minimizing the number of phases. The greedy algorithm focuses on balancing the load and reducing network contention while the all-to-all based algorithm considers all three issues when scheduling messages. Details about these algorithms can be found in [12].

### Algorithms for MPI\_Allgatherv

Most of the topology unaware all-gather algorithms are extended to the all-gather operation. The algorithms include the **simple**, **recursive doubling**, **ring** and **pair** families algorithms. The topology specific algorithm is based on the topology specific all-gather algorithm.

## 3.2 Timing mechanisms

The timing mechanisms constitute the most critical component in the tuning system as it decides how the performance of a routine is measured. Since the measurement results are used to select the best algorithms, it is essential that the timing mechanism gives accurate timing results. Unfortunately, the performance of a communication routine depends largely on the application behavior. Our system makes the timing module extensible in addition to providing some built-in timing mechanisms. This allows users to supply application specific timing mechanisms that can closely reflect the application behavior. The built-in timing mechanisms follow the approach in Mppctest [8], where the performance of multiple invocations of a routine is measured, and the timing results are fairly consistent and repeatable for the routines we currently support. We plan to add more built-in timing mechanisms as we consider other MPI routines.

## 3.3 Search heuristics

The search heuristics decide the order that the parameter space is searched to find the best algorithm parameters, which decide the time to tune a routine. In the current system, only the ring with  $N$  MPI barriers and pair with  $N$  MPI barriers algorithms have one algorithm parameter,  $N\_mpi\_barrier$ , which has a small solution space. Our current system only supports a linear search algorithm, that is, deciding the best solution for each parameter by linearly trying out all potential values. The system handles multiple parameters cases by assuming that the parameters are independent from each other. The linear search algorithm is sufficient for our current system. In the future, as we consider other MPI operations and add algorithms with parameters that have larger solution spaces, we will add more efficient search heuristics, such as hill climbing.

## 3.4 Drivers

The process for tuning *MPI\_Alltoall*, *MPI\_Allgather*, and *MPI\_Allreduce* is different from that for tuning *MPI\_Alltoallv* and *MPI\_Allgatherv*, and that is depicted in Figure 2. The process contains four steps. In the first step, the system prompts the user for inputs, which include (1) the routine(s) to tune, (2) whether to consider topology specific routines, (3) which routine generator to use (users can choose the built-in generator or supply their own generator), (4) the topology description file, (5) the timing mechanism (users can choose among the built-in ones or supply their own timing program). In the second step, the system generates the topology specific routines if requested. In the third step, algorithms with parameters are tuned. The tuning is carried out as follows. First, for a set of fixed message sizes (currently set to  $1B$ ,  $64B$ ,  $512B$ ,  $1KB$ ,  $2KB$ ,  $4KB$ ,  $8KB$ ,  $16KB$ ,  $32KB$ ,  $64KB$ ,  $128KB$ , and  $256KB$ ), the linear search algorithm is used to find the optimal value for each parameter for each of the sizes. The system then walks through each pair of adjacent message sizes. If the best parameter values are the same for the two sizes, the system

will use the parameter values for all message sizes in the range between the two sizes. If different best parameter values are used for the two points of the message sizes, a binary search algorithm is used to decide the crossing point where the parameter value should be changed. For each operation, the system assumes the same algorithm when the message size is larger than or equal to  $256KB$ . This step generates a tuned routine for the particular algorithm with the best parameter values set for different ranges of message sizes. This tuned routine is stored back in the algorithm repository as an algorithm without parameters. In the last step, all algorithms with no parameters are considered. The process is similar to that in step 3. The only difference is that instead of tuning an algorithm with different parameter values, this step considers different algorithms. Figure 3 (a) shows an example of the final generated all-to-all routine.

Step 1: Prompt the user for the following information:

- 1.1 Which routine to tune;
- 1.2 Whether to include topology specific routines;
- 1.3 Which routine generator to use;
- 1.4 The topology description file;
- 1.5 Which timing mechanism to use;

Step 2: Generate the topology specific routines.

Step 3: Tune algorithms with parameters.

- 3.1 Decide the best par. values for a set of msg sizes (currently 1B, 64B, 256B, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB).
- 3.2 Find the exact message sizes when the best parameter values are changed (binary search).
- 3.3 Generate one routine with the best parameters set and store it in the algorithm repository.

Step 4: final tuning, generate the final routine.

```
/* only considers algorithms with no parameters */
4.1 Decide the best algorithm for a set of message sizes (currently 1B, 64B, 256B, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB).
4.2 Find the exact message sizes when the best algorithms are changed using (binary search).
4.3 Generate the final routine with the best algorithms selected for different message ranges.
```

**Figure 2: A tuning process example**

When tuning *MPI\_Alltoallv* and *MPI\_Allgatherv*, the system also asks for the pattern description file in addition to other information. The routine generator uses both the topology and pattern information and produces a routine for the specific topology and pattern. Tuning algorithms with parameters in the third step is straight-forward, the system just measures the performance of all potential values for a parameter for the specific pattern and decides the best parameter values. Finally, the last step considers all algorithms and selects the best algorithm. The system potentially generates a different implementation for each invocation of a routine. To produce a compact routine for an application, the system allows the pattern description file to

```
int alltoall_tuned(...) {
  if ((msg_size >= 1) && (msg_size < 8718))
    alltoall_simple(...);
  else if ((msg_size >= 8718) && (msg_size < 31718))
    alltoall_pair_light_barrier(...);
  else if ((msg_size >= 31718) && (msg_size < 72032))
    alltoall_pair_N_mpi_barrier_tuned(...);
  else if (msg_size >= 72032)
    alltoall_pair_mpi_barrier(sbuff, scout, ...);
}
```

(a) An example tuned MPI\_Alltoall routine

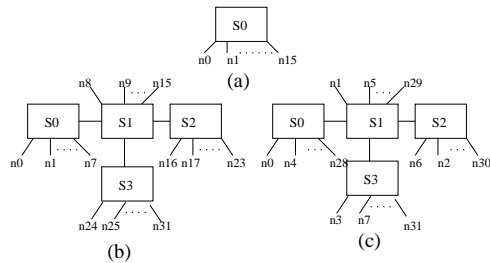
```
int alltoallv_tuned(...) {
  static int pattern = 0;
  if (pattern == 0) {
    alltoallv_tspecific_alltoall(...); pattern++;
  } else if ((pattern >= 1) && (pattern < 100)) {
    alltoallv_ring(...); pattern++;
  } else { MPI_alltoallv(...); pattern++; }
}
```

(b) An example tuned MPI\_Alltoallv routine

**Figure 3: Examples of tuned routines**

contain multiple patterns, which may correspond to the sequence of invocations of the routine in the application. This pattern file can be created by profiling the program execution. The tuning system then creates a sequence of tuned implementations for the sequence of patterns. To reduce the code size, before a tuned routine is generated for a pattern, the pattern is compared with other patterns whose routines have been generated. If the difference is under a threshold value, the old tuned routine will be used for the new pattern. Figure 3 (b) shows an example of the final generated all-to-allv routine for an application.

## 4. EXPERIMENTS



**Figure 4: Topologies used in the experiments**

The experiments are performed on Ethernet-switched clusters. The nodes in the clusters are Dell Dimension 2400 with a 2.8GHz P4 processor, 128MB of memory, and 40GHz of disk space. All machines run Linux (Fedora) with the 2.6.5-1.358 kernel. The Ethernet card in each machine is Broadcom BCM 5705 with the driver from Broadcom. These machines are connected to Dell Powerconnect 2224 and Dell Powerconnect 2324 100Mbps Ethernet switches.

Topo.	MPI_Alltoall	MPI_Allgather	MPI_Allreduce
(a)	Simple ( $n < 8718$ ) Pair light barrier ( $n < 31718$ ) Pair N MPI barriers ( $n < 72032$ ) Pair MPI barrier (else)	2D Mesh ( $n < 10844$ ) Topo. specific LR ( $n < 75968$ ) Pair MPI barrier (else)	Tuned all-gather ( $n < 112$ ) Rdb ( $n < 9468$ ) Rab. variation 1 ( $n < 60032$ ) MPICH Rab. ( $n < 159468$ ) Rab. variation 2 (else)
(b)	Bruck ( $n < 112$ ) Ring ( $n < 3844$ ) Ring N MPI barriers ( $n < 6532$ ) Ring light barrier ( $n < 9968$ ) Pair N MPI barriers ( $n < 68032$ ) Pair MPI barrier (else)	3D Mesh ( $1n < 208$ ) Topo. specific LR (else)	Rab. variation 1 ( $n < 17$ ) Rdb ( $n < 395$ ) MPICH Rab. ( $n < 81094$ ) Rab. variation 2 (else)
(c)	Bruck ( $n < 86$ ) Simple ( $n < 14251$ ) Pair MPI barrier (else)	3D Mesh ( $n < 3999$ ) Topo. specific LR (else)	Tuned all-gather ( $n < 17$ ) Rdb ( $n < 489$ ) Rab. variation 1 ( $n < 20218$ ) Rab. variation 2 (else)

**Table 1: Tuned MPI\_Alltoall, MPI\_Allgather, and MPI\_Allreduce**

We conducted experiments on many topologies. In all experiments, the tuned routines are robust and offer high performance. Due to space limitation, we will report selected results on three representative topologies, which are shown in Figure 4. Figure 4 (a) is a 16-node cluster connected by a single 2324 switch. Parts (b) and (c) of the figure show 32-node clusters of different logical topologies but the same physical topology, each having four switches with 8 nodes attached. Most current MPI implementations use a naive logical to physical topology mapping scheme. We will refer to these three topologies as *topology (a)*, *topology (b)*, and *topology (c)*.

We compare the performance of our tuned routines with routines in LAM/MPI 6.5.9 and a recently improved MPICH 1.2.6 [21] using both micro-benchmarks and applications. The tuned routines are built on LAM point-to-point primitives. To make a fair comparison, we port MPICH-1.2.6 routines to LAM. We will use MPICH-LAM to represent the ported routines. We will use the term TUNED to denote the tuned routines. In the evaluation, TUNED is compared with native LAM, native MPICH, and MPICH-LAM.

#### 4.1 Tuned routines and tuning time

Table 1 shows the tuned *MPI\_Alltoall*, *MPI\_Allgather*, and *MPI\_Allreduce* for topologies (a), (b), and (c). In this table, the algorithms selected in the tuned routine are sorted in the increasing order based on their applicability to the message sizes. For comparison, the algorithms in LAM/MPI and MPICH are depicted in Table 2. Since topologies (a), (b), and (c) have either 16 nodes or 32 nodes, only the algorithms for 16 nodes or 32 nodes are included in Table 2. There are a number of important observations. First, from Table 1, we can see that for different topologies, the optimal algorithms for each operation are quite different, which indicates that the one-scheme-fits-all approach in MPICH and LAM cannot achieve good performance for different topologies. Second, the topology specific algorithms are part of the tuned *MPI\_Allgather* and *MPI\_Allreduce* routines for all three topologies. Although the topology specific all-to-all routine is not selected in the tuned routines for the three topologies, it offers the best performance for other topologies when the message size is large. These indicate that using

topology unaware algorithms alone is insufficient to obtain high performance routines. Hence, an empirical approach must be used with the topology specific routines to construct efficient communication routines for different topologies. Third, although the MPICH algorithms in general are much better than LAM algorithms, in many cases, they do not use the best algorithms for the particular topology and for the particular message size. As will be shown in the next subsection, by empirically selecting better algorithms, our tuned routines sometimes out-perform MPICH routines to a very large degree.

routine	LAM	MPICH
<i>Alltoall</i>	Simple	Bruck ( $n \leq 256$ ) Spreading Simple ( $n \leq 32768$ ) Pair (else)
<i>Allgather</i>	Gather-bcast	Rdb ( $n * p < 524288$ ) Topo. unaware LR (else)
<i>Allreduce</i>	Reduce-bcast	Rdb ( $n < 2048$ ) MPICH Rab. (else)

**Table 2: LAM/MPI and MPICH algorithms for MPI\_Alltoall, MPI\_Allgather, and MPI\_Allreduce**

tuned routines	topo. (a)	topo. (b)	topo. (c)
<i>MPI_Alltoall</i>	1040s	6298s	6295s
<i>MPI_Allgather</i>	1157s	6288s	6326s
<i>MPI_Allreduce</i>	311s	261s	296s
<i>MPI_Alltoallv</i>	64s	177s	149s
<i>MPI_Allgatherv</i>	63s	101s	112s

**Table 3: Tuning time (seconds)**

Table 3 shows the tuning time of our current system. In the table, the tuning time for *MPI\_Allreduce* assumes that *MPI\_Alltoall* and *MPI\_Allgather* have been tuned. The time for *MPI\_Alltoallv* is the tuning time for finding the best routine for one communication pattern: all-to-all with 1KB message size. The time for *MPI\_Allgatherv* is the tuning time for finding the best routine for one communication pattern: all-gather with 1KB message size. The tuning time depends on many factors such as the number of algorithms

to be considered, the number of algorithms having parameters and the parameter space, the search heuristics, the network topology, and how the timing results are measured. As can be seen from the table, it takes minutes to hours to tune the routines. The time is in par with that for other empirical approach based systems such as ATLAS [23]. Hence, like other empirical based systems, our tuning system can apply when this tuning time is relatively insignificant, e.g. when the application has a long execution time, or when the application is executed repeatedly on the same system.

## 4.2 Performance of individual routine

```
MPIBarrier(MPLCOMM_WORLD);
start = MPI_Wtime();
for (count = 0; count < ITER_NUM; count++) {
    MPIAlltoall(...);
}
elapsed_time = MPI_Wtime() - start;
```

**Figure 5: Code segment for measuring the performance of an individual MPI routine.**

We use an approach similar to Mpptest [8] to measure the performance of an individual MPI routine. Figure 5 shows an example code segment for measuring the performance. The number of iterations is varied according to the message size: more iterations are used for small message sizes to offset the clock inaccuracy. For the message ranges 1B-3KB, 4KB-12KB, 16KB-96KB, 128KB-384KB, and 512KB, we use 100, 50, 20, 10, and 5 iterations, respectively. The results for these micro-benchmarks are the averages of three executions. We use the *average* time among all nodes as the performance metric.

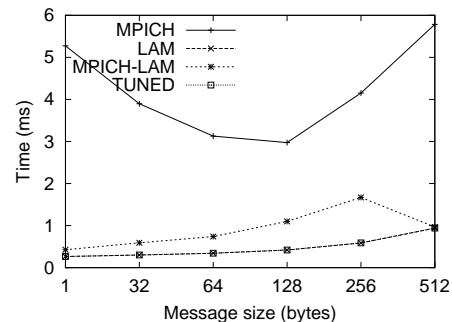
We will report results for *MPI\_Alltoall*, *MPI\_Allgather*, and *MPI\_Allreduce*. The performance of *MPI\_Alltoallv* and *MPI\_Allgatherv* depends on the communication pattern. The two routines will be evaluated with applications in the next subsection. Since in most cases, MPICH has better algorithms than LAM, and MPICH-LAM offers the highest performance among systems we compare to, we will focus on comparing TUNED with MPICH-LAM. Before we present the selected results, we will point out two general observations in the experiments.

1. Ignoring the minor inaccuracy in performance measurement, for all three topologies and all three operations, the tuned routines never perform worse than the best corresponding routines in LAM, MPICH, and MPICH-LAM.
2. For all three topologies and all three operations, the tuned routines out-perform the best corresponding routines in LAM, MPICH, and MPICH-LAM by at least 40% at some ranges of message sizes.

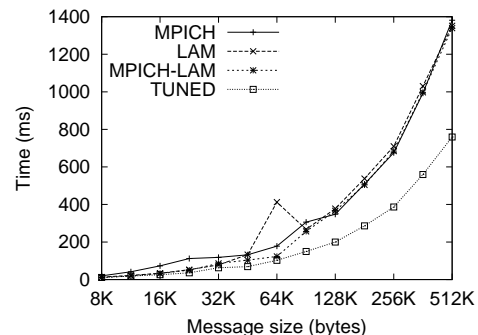
Figure 6 shows the performance of *MPI\_Alltoall* results on topology (a). For small messages ( $1 \leq n \leq 256$ ), both LAM and TUNED use the simple algorithm, which offers higher performance than the Bruck algorithm used in MPICH. When the message size is 512 bytes, MPICH changes to the spreading simple algorithm, which has similar performance to the simple algorithm. TUNED, LAM, and MPICH-LAM

have similar performance for the message size in the range from 256 bytes to 9K bytes. Figure 6 (b) shows the results for larger message sizes. For large messages, TUNED offers much higher performance than both MPICH and LAM. For example, when the message size is 128KB, the time for TUNED is 200.1ms and the time for MPICH-LAM (the best among LAM, MPICH, and MPICH-LAM) is 366.2ms, which constitutes an 83% speedup. The performance curves for topology (b) and topology (c) show a similar trend. Figure 7 shows the results for topology (b). For a very wide range of message sizes, TUNED is around 20% to 42% better than the best among LAM, MPICH, and MPICH-LAM.

Figure 8 shows the performance results for *MPI\_Allgather* on topology (c). When the message size is small, TUNED performs slightly better than other libraries. However, when the message size is large, the tuned routine significantly outperforms routines in other libraries. For example, when the message size is 32KB, the time is 102.5ms for TUNED, 1362ms for LAM, 834.9ms for MPICH, and 807.9ms for MPICH-LAM. TUNED is about 8 times faster than MPICH-LAM. This demonstrates how much performance differences it can make when the topology information is taken into consideration. In fact, the topology specific logical ring algorithm, used in TUNED, in theory can achieve the same performance for any Ethernet switched cluster with any number of nodes as the performance for a cluster with the same number of nodes connected by a single switch. On the other hand, the performance of the topology unaware logical ring algorithm, used in MPICH, can be significantly affected by the way the logical nodes are organized.



(a) Small message sizes

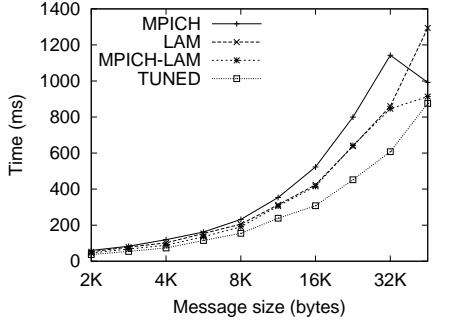


(b) Medium to large message sizes

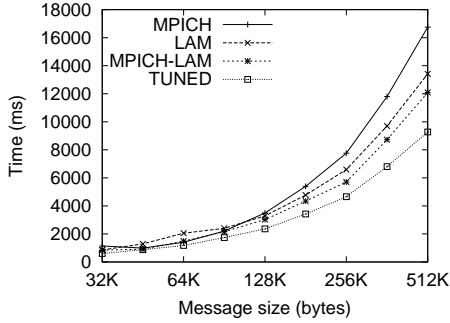
**Figure 6: *MPI\_Alltoall* on topology (a)**

Figure 9 shows the results for *MPI\_Allreduce* on topology (c). TUNED and MPICH-LAM have a similar performance when the message size is less than 489 bytes. When the



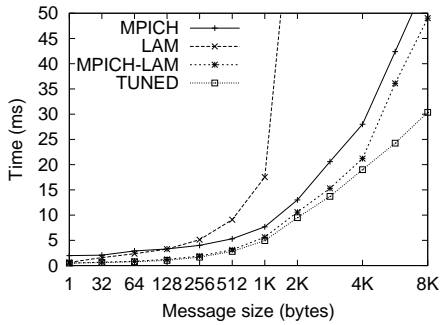


(a) Medium message sizes

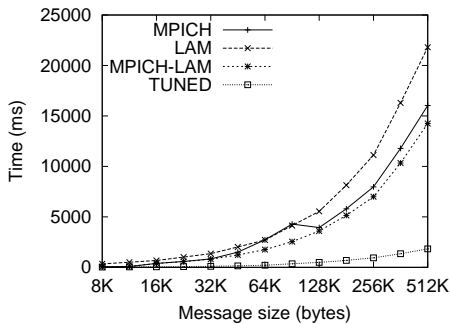


(b) Large message sizes

Figure 7: *MPI\_Alltoall* on topology (b)

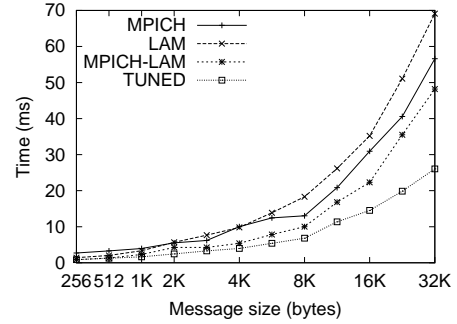


(a) Small message sizes

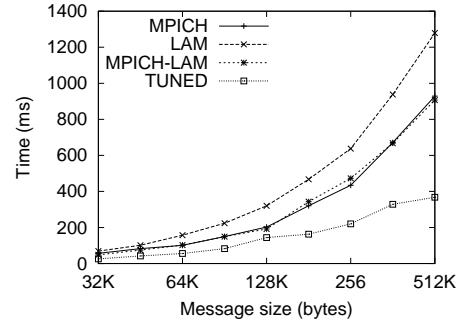


(b) Medium to large message sizes

Figure 8: *MPI\_Allgather* on topology (c)



(a) Small to medium message sizes



(b) large message sizes

Figure 9: *MPI\_Allreduce* on topology (c)

message size is larger, TUNED outperforms MPICH-LAM to a very large degree even though for a large range of message sizes, both TUNED and MPICH-LAM use variations of the Rabenseifner algorithm. For example, for message size 2048 bytes, the time is 2.5ms for TUNED versus 4.3 ms for MPICH-LAM. For message size 64KB, the time is 55.9ms for TUNED versus 102.4ms for MPICH-LAM.

### 4.3 Performance of application programs

We use three application programs in the evaluation: IS, FT, and NTUB. IS and FT come from the *Nas Parallel Benchmarks* NPB [5]. The IS (Integer Sort) benchmark sorts  $N$  keys in parallel and the FT (Fast Fourier Transform) benchmark solves a partial differential equation (PDE) using forward and inverse FFTs. Both IS and FT are communication intensive programs with most communications performed by *MPI\_Alltoall* and *MPI\_Alltoallv* routines. We use the class B problem size supplied by the benchmark suite for the evaluation. The NTUB (Nanotube) program performs molecular dynamics calculations of thermal properties of diamond [18]. The program simulates 1600 atoms for 1000 steps. This is also a communication intensive program with most communications performed by *MPI\_Allgather*.

Table 4 shows the execution time for using different libraries with different topologies. The tuned library consistently achieves much better performance than the other implementations for all three topologies and for all programs. For example, on topology (a), TUNED improves the IS performance by 59.8% against LAM, 338.1% against MPICH, and 61.9% against MPICH-LAM. Notice that the execution time on topologies (b) and (c) is larger than that on topology (a) even though there are 32 nodes on topologies (b) and (c) and 16 nodes on topology (a). This is because all

	library	topo. (a)	topo. (b)	topo. (c)
I S	LAM	15.5s	38.4s	36.5s
	MPICH	42.5s	58.2s	51.5s
	MPICH-LAM	15.7s	35.5s	33.4s
	TUNED	9.7s	28.4s	28.6s
F T	LAM	409.4s	320.8s	281.4s
	MPICH	243.3s	365.8s	281.1s
	MPICH-LAM	242.0s	246.0s	305.6s
	TUNED	197.7s	206.0s	209.8s
N U	LAM	214.3s	304.1s	179.6s
	MPICH	49.7s	244.5s	88.7s
T B	MPICH-LAM	47.2s	236.8s	80.9s
	TUNED	35.8s	47.6s	45.0s

**Table 4: Execution time (seconds)**

programs are communication bounded and the network in topologies (b) and (c) has a smaller aggregate throughput than that in topology (a).

## 5. CONCLUSION

In this paper, we present an automatic generation and tuning system for MPI collective communication routines. By integrating the architecture specific information with an empirical approach, the system is able to produce very efficient routines that are customized for the specific platform. The experimental results confirm that the tuned routines out-perform existing MPI libraries to a very large degree. We are currently extending the system to produce other MPI collective communication routines and exploring various timers so that more accurate timing results can be used to guide the tuning process.

## 6. REFERENCES

- [1] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing Matrix Multiply using PHIPAC: a Portable, High-Performance, ANSI C Coding Methodology. In *Proceedings of the ACM SIGARC International Conference on SuperComputing*, 1997.
- [2] J. Bruck, C. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-all Communications in Multiport Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143-1156, Nov. 1997.
- [3] A. Faraj, P. Patarasuk, and X. Yuan. Bandwidth Efficient All-to-All Broadcast on Switched Clusters. *Technical Report*, Department of Computer Science, Florida State University, May 2005.
- [4] A. Faraj and X. Yuan. Message Scheduling for All-to-all Personalized Communication on Ethernet Switched Clusters. *IEEE IPDPS*, April 2005.
- [5] NASA Parallel Benchmarks. Available at <http://www.nas.nasa.gov/NAS/NPB>.
- [6] M. Frigo and S. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, page 1381, 1998.
- [7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. In *MPI Developers Conference*, 1995.
- [8] W. Gropp and E. Lusk. Reproducible Measurements of MPI Performance Characteristics. *Technical Report ANL/MCS-P755-0699*, Argonne National Laboratory, Argonne, IL, June 1999.
- [9] LAM/MPI Parallel Computing. <http://www.lam-mpi.org>.
- [10] M. Lauria and A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1), January 1997.
- [11] L. V. Kale, S. Kumar, K. Varadarajan, "A Framework for Collective Personalized Communication," *IPDPS'03*, April 2003.
- [12] A. Karwande, X. Yuan, and D. K. Lowenthal. CC-MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters. In *ACM SIGPLAN PPOPP*, pages 95-106, June 2003.
- [13] T. Kielmann, et. al. Magpie: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *ACM SIGPLAN PPOPP*, pages 131-140, May 1999.
- [14] R. Rabenseifner. A new optimized MPI reduce and allreduce algorithms. Available at <http://www.hlrs.de/organization/par/services/models/mpi/myreduce.html>, 1997.
- [15] The MPI Forum. *The MPI-2: Extensions to the Message Passing Interface*, July 1997. Available at <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>.
- [16] MPICH - A Portable Implementation of MPI. <http://www.mcs.anl.gov/mpi/mpich>.
- [17] H. Ogawa and S. Matsuoka. OMPI: Optimizing MPI Programs Using Partial Evaluation. In *Supercomputing'96*, November 1996.
- [18] I. Rosenblum, J. Adler, and S. Brandon. Multi-processor molecular dynamics using the Brenner potential: Parallelization of an implicit multi-body potential. *International Journal of Modern Physics, C* 10(1):189-203, Feb. 1999.
- [19] S. Sistare, R. vandeVaart, and E. Loh. Optimization of MPI Collectives on Clusters of Large Scape SMPs. In *Proceedings of SC99: High Performance Networking and Computing*, 1999.
- [20] H. Tang, K. Shen, and T. Yang. Program Transformation and Runtime Support for Threaded MPI Execution on Shared-Memory Machines. *ACM Transactions on Programming Languages and Systems*, 22(4):673-700, July 2000.
- [21] R. Thakur, R. Rabenseifner, and W. Gropp. Optimizing of Collective Communication Operations in MPICH. *ANL/MCS-P1140-0304*, Mathematics and Computer Science Division, Argonne National Laboratory, March 2004.
- [22] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically Tuned Collective Communications. In *Proceedings of SC'00: High Performance Networking and Computing*, 2000.
- [23] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing'98: High Performance Networking and Computing*, 1998.