STAR-MPI: Self Tuned Adaptive Routines for MPI Collective Operations

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

David Lowenthal
Department of Computer Science
University of Georgia
Athens, GA 30602
dkl@cs.uga.edu

ABSTRACT

Message Passing Interface (MPI) collective communication routines are widely used in parallel applications. In order for a collective communication routine to achieve high performance for different applications on different platforms, it must be adaptable to *both* the system architecture and the application workload. Current MPI implementations do not support such software adaptability and are not able to achieve high performance on many platforms. In this paper, we present STAR-MPI (**S**elf **T**uned **A**daptive **R**outines for **MPI** collective operations), a set of MPI collective communication routines that are capable of adapting to system architecture and application workload. For each operation, STAR-MPI maintains a set of communication algorithms that can potentially be efficient at different situations. As an application executes, a STAR-MPI routine applies the Automatic Empirical Optimization of Software (AEOS) technique at run time to dynamically select the best performing algorithm for the application on the platform. We describe the techniques used in STAR-MPI, analyze STAR-MPI overheads, and evaluate the performance of STAR-MPI with applications and benchmarks. The results of our study indicate that STAR-MPI is robust and efficient. It is able to find efficient algorithms with reasonable overheads, and it out-performs traditional MPI implementations to a large degree in many cases.

1. INTRODUCTION

The standardization of Message Passing Interface (MPI) [14] has facilitated the development of scientific parallel applications using explicit message passing as the programming paradigm and has resulted in a large number of MPI based parallel applications. For these applications to achieve high performance, it is crucial that an MPI library realizes the communications efficiently.

Studies have indicated that MPI collective operations are used in most MPI applications and account for a significant portion of the total time in some applications [21]. Developing efficient MPI collective communication routines, however, is challenging. For collective communication routines to achieve high performance for different applications on different platforms, they must be able to adapt to the system/application configuration: the most efficient algorithm for a given configuration may be vastly different from that for a different configuration. The challenges in developing efficient MPI collective communication routines lie not so much in the development of an individual algorithm for a given situation. In fact, many efficient algorithms for various operations have been developed for different networks and different topologies. The main challenge lies in the mechanism for a library routine to adapt to system/application configurations and to find the most efficient algorithm for a given configuration.

The current paradigm for implementing MPI libraries is that for a given operation, the library developer must decide at library design time which algorithm is to be used in a given situation (such as a given message size, a given type of network, etc). MPI libraries developed using this paradigm such as MPICH [15] and LAM/MPI [9] can only support a limited form of software adaptability and cannot achieve high performance on many platforms for the following reasons [3]. First, since the algorithms are decided before the system is known, the algorithms cannot be optimized for the system parameters. Many system parameters, including network topology, nodal architecture, context switching overheads, ratio between the network and the processor speeds, the switch design, and the amount of buffer memory in switches, can significantly affect the performance of a communication algorithm. It is impossible for the library developer to make the right choices for different platforms. Second, the application behavior, which can also significantly affect the performance, cannot be taken into consideration in the development of the library.

We propose a technique, which we call *delayed finalization of MPI collective communication routines* (DF), to improve the software adaptability in MPI libraries. The idea is to postpone the decision of which algorithm to use for a collective operation until after the platform and/or the application are known. This potentially allows architecture and/or application specific optimizations to be applied. A DF system has two major components: (1) an algorithm repository that contains, for each operation, an extensive set of topology/system unaware and topology/system specific algorithms that can potentially achieve high performance in different situations, and (2) an automatic algorithm selection mechanism to determine the algorithms to be used in the final routine. The DF library developers only implement the communication algorithms and the mechanisms to select the algorithms, but do not make decisions about which algorithms to use in an operation. The final algorithms for an operation are automatically selected by the algorithm selection mechanism, which may take system architecture and application into account.

ICS'06, June 28–30, Cairns, Queensland, Australia.

In this paper, we present a prototype DF system named STAR-MPI (**S**elf **T**uned **A**daptive **R**outines for **MPI** collective operations). STAR-MPI maintains a set of algorithms for each operation and applies the Automatic Empirical Optimization of Software (AEOS) technique [27] at run time to dynamically select (tune) the algorithms as the application executes. STAR-MPI targets programs that invoke a collective routine a large number of times (programs that run for a large number of iterations). One major issue in STAR-MPI is whether the AEOS technique can effectively select good algorithms at run time. Hence, our primary objective is to develop AEOS techniques that can find the efficient algorithms at run time. Under the condition that efficient algorithms can be found, the secondary objective is to reduce the tuning overhead. STAR-MPI incorporates various techniques for reducing the tuning overhead while selecting an efficient algorithm. We describe the techniques used in STAR-MPI, study the tuning overheads, and evaluate the performance of STAR-MPI using benchmarks and application programs. The results of our study show that (1) STAR-MPI is robust and effective in finding efficient MPI collective routines; (2) the tuning overheads are manageable when the message size is reasonably large; and (3) STAR-MPI finds the efficient algorithms for the particular platform and application, which not only out-perform traditional MPI implementations to a large degree in many cases, but also offers better performance in many cases than a static tuning system [3] with a super-set of algorithms.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes the STAR-MPI library. Section 4 reports the results of the performance study, 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[15, 25] and LAM/MPI [9]. Many researchers have worked on optimizing the MPI library [10, 11, 12, 16, 23, 24, 25]. In [12], optimizations are proposed for collective communications over Wide-Area Networks by considering the network details. In [16], 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 [24]. Optimizations for clusters of SMPs are presented in [23]. A combined compiler and library approach was proposed in [11]. There is a large body of work on the development of algorithms for MPI collective operations [4, 5, 18, 25]. This work, however, does not focus on developing collective communication algorithms. It is concerned about how to achieve software adaptability by selecting the best algorithm among a set of communication algorithms.

The AEOS technique has been applied successfully to various computational library routines [1, 6, 27]. Using the empirical approach to tune MPI collective algorithms was proposed in [26] and later extended in [3]. The systems in [26, 3] tune the routines statically at the library installation time. While these systems delay the finalization of MPI routines until the platform is known, they have limitations. First, the algorithm selection process, also called tuning process, lasts for a long time since each algorithm must be executed and measured for many message sizes. This limits the applicability of the system: it is beneficial only when the platform is fixed and the routines are repeatedly invoked. Second, since the routines are tuned using some standard performance measurement scheme (e.g. Mpptest [7]), the tuned routines may not select the most efficient algorithm for a given application.

While both STAR-MPI and our previous static tuning system [3] employ the AEOS approach to select the best performing algorithm from a repository of communication algorithms, apart from using a similar set of communication algorithms, STAR-MPI is fundamentally different from static tuning. First, the major research issue in STAR-MPI is (1) whether it is possible to find good algorithms dynamically at run time and (2) how to find the algorithms with minimum tuning. This is not a problem in the static tuning system. Second, the techniques in STAR-MPI can be directly applied to the implementation of a traditional MPI library while the static tuning system produces a different type of library (users must go through the static tuning to obtain the final routine). Third, STAR-MPI is different from the static tuning system in terms of applicability. STAR-MPI is effective on a typical supercomputing cluster where users get different partitions, and therefore potentially different network characteristics, every time they run a job. Static tuning in such an environment can be prohibitively expensive. Finally, STAR-MPI measures the performance of each algorithm in the context of application execution, which results in a more accurate measurement. Our performance study indicates that STAR-MPI often selects better algorithms than our previous static tuning system [3], which in turn usually selects more efficient algorithms than the ones in traditional MPI implementations. We are not aware of any MPI implementations similar to STAR-MPI.

3. STAR-MPI

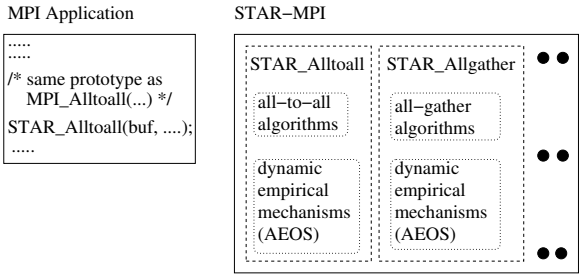


Figure 1. High level view of STAR-MPI.

The high level view of the STAR-MPI library is shown in Figure 1. STAR-MPI is an independent layer or library that contains a set of collective communication routines whose prototypes are the same as the corresponding MPI collective communication routines. MPI programs can be linked with STAR-MPI to accesses the adaptive routines. As shown in the figure, an *MPI_Alltoall* in an application is replaced with *STAR_Alltoall* routine. Different from traditional MPI implementations, each STAR-MPI routine has access to an algorithm repository that contains multiple implementations for the operation. In addition, each STAR-MPI routine incorporates a dynamic Automatic Empirical Optimization of Software (AEOS) [27] module, which performs self-monitoring and self-tuning during program execution. By maintaining multiple algorithms that can achieve high performance in different situations for each operation and using a dynamic empirical approach to select the most efficient algorithm, STAR-MPI is able to adapt to the application and platform.

STAR-MPI runs over MPICH. The routines supported in STAR-MPI include *MPI_Alltoall*, *MPI_Allgather*, *MPI_Allgatherv*, *MPI_Allreduce*, and *MPI_Bcast*. STAR-MPI is designed for Ethernet switched clusters. All algorithms in STAR-MPI come from the algorithm repository of our static tuning system [3], which was designed for Ethernet switched clusters. Hence, STAR-MPI achieves the best results on Ethernet switched clusters, although it can also tune routines for other types of clusters. In the following, we will first summarize briefly the algorithms maintained in STAR-MPI and then describe the dynamic AEOS technique.

3.1 COLLECTIVE ALGORITHMS IN STAR-MPI

As shown in Figure 1, each collective routine in STAR-MPI includes an algorithm repository that contains a set of communication algorithms that can potentially achieve high performance in different situations. The organization of the algorithm repository is similar to that in [3]. It includes both topology unaware algorithms and topology specific algorithms. The topology specific algorithms are automatically generated based on the topology information when it is available. In cases when the topology information is not available, only the topology unaware algorithms are used in the tuning. Meta-data are associated with each algorithm to describe important properties of the algorithms. One example of the meta-data is the range of the message sizes where the algorithm can be efficiently applied. The AEOS algorithm in STAR-MPI may use the meta-data information to decide the communication algorithms that would be included in a particular tuning process.

Selecting algorithms to be included in STAR-MPI is very important for the performance since (1) the number of algorithms in the repository directly affects the time to tune the routine, and (2) the performance of each of the algorithms selected directly affects the program execution time since it must be executed even if it is not used in the final routine. Hence, the criteria for including an algorithm in STAR-MPI are more strict than those for including an algorithm in the static tuning system: the set of algorithms used in STAR-MPI is a subset of the algorithms in our static tuning system [3]. The selection represents a trade-off between overhead and effectiveness. Including more algorithms makes STAR-MPI more effective and robust, but will introduce more overhead in the tuning process. Our selection is based on our experience with the static tuning system. First, we remove algorithms that are rarely selected by the static tuning system. Second, some algorithms in the static tuning system have a large parameter space. It takes a long tuning time to obtain the algorithm with the best parameter values. STAR-MPI replaces such an algorithm (with a large parameter space) by a small number of most promising algorithm instances. Next, we will briefly describe the STAR-MPI algorithms for *MPI_Alltoall*, *MPI_Allgather*, and *MPI_Allreduce*. These routines are used in the evaluation. *MPI_Allgatherv* has exactly the same sets of algorithms as *MPI_Allgather*.

Algorithms for MPI_Alltoall

There are 13 all-to-all algorithms in STAR-MPI. Four of the algorithms are only for small messages (message size less than or equal to 256 bytes). The description of the algorithms follows. In the description, $i \rightarrow j$ denotes the communication from node i to node j and p is the number of processes.

Simple. This algorithm basically posts all receives and all sends, starts the communications, and waits for all communications to finish. The order of communications for node i is $i \rightarrow i+1$, $i \rightarrow i+2$, ..., $i \rightarrow (i+p-1)\ mod\ p$.

2D mesh. 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.

3D mesh. This algorithm extends the 2D mesh algorithm to a 3D mesh algorithm by organizing the nodes as a logical $x \times y \times z$ mesh.

Recursive doubling (rdb). When the number of processes is a power of two, the recursive doubling algorithm is 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 [25].

Bruck. 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, 25].

The *2D mesh*, *3D mesh*, *rdb*, and *Bruck* algorithms are designed for small messages. STAR-MPI only uses them to tune for messages up to 256 bytes.

Ring. This algorithm partitions the all-to-all communication into $p-1$ steps (phases). In step i, node j sends a messages to node $(j+i)\ mod\ p$ and receives a message from node $(j-i)\ mod\ p$. 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).

Ring with light barrier. This algorithm adds light-weight barriers between the communications in different phases that can potentially cause node contention and eliminates such contention.

Ring with MPI barrier. 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.

Pair. 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). 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*, and *pair with MPI barrier* algorithms.

Topology specific. 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 (1) communications within each phase do not have contention, and (2) a minimum number of phases are used to complete the communication. STAR-MPI has two variations of the topology specific algorithm with different synchronization mechanisms.

Algorithms for MPI_Allgather

STAR-MPI maintains 12 algorithms for *MPI_Allgather*. The all-gather communication pattern is a special all–to–all communication pattern (sending the same copy of data to each node instead of sending different messages to different nodes). The STAR-MPI algorithm repository for *MPI_Allgather* includes the following algorithms that work similar to their all-to-all counterparts: *simple*, *2D mesh*, *3D mesh*, *rdb*, *ring*, *ring with light barrier*, *ring with MPI barrier*, *pair*, *pair with light barrier*, *pair with MPI barrier*. STAR-MPI also includes the Bruck all-gather algorithm [2], which is different from the Bruck all-to-all algorithm. Details can be found in [2, 25]. In addition, the STAR-MPI repository includes a topology-specific logical ring (TSLR) algorithm [5]. Let n'_0, n'_1, ..., n'_{p-1} be a permutation of the p processors. TSLR finds a permutation such that the communications in $n'_0 \rightarrow n'_1 \rightarrow ... \rightarrow n'_{p-1} \rightarrow n'_0$ do not cause network contention. TSLR realizes the all-gather operation by performing the $n'_0 \rightarrow n'_1 \rightarrow ... \rightarrow n'_{p-1} \rightarrow n'_0$ pattern $p-1$ times.

Algorithms for MPI_Allreduce

STAR-MPI maintains 20 algorithms for *MPI_Allreduce*.
These algorithms can be classified into three types.

In the first type of algorithms, the *MPI_Allreduce* operation is performed by first using an *MPI_Allgather* to gather the data in all nodes and then performing the reduction operation. The all-gather

has the following variations: *Bruck, 2D mesh, 3D mesh, rdb,* and *ring*.

The second-type algorithms are variations of the Rabenseifner algorithm [20], where the all-reduce operation is performed by a reduce-scatter operation followed by an all-gather operation. The reduce-scatter is realized by recursive halving [20] and the all-gather implementation has the following variations: *simple*, *2D mesh*, *3D mesh*, *rdb*, *Bruck*, *ring*, *ring with light barrier*, *ring with MPI barrier*, and *TSLR*. We will use *Rab1-x* to denote this type of algorithms with x all-gather implementations. For example, *Rab1-2D* means the variation with the *2D mesh* all-gather algorithm.

The third-type algorithms are also variations of the Rabenseifner algorithm [20], where the all-reduce operation is performed by a reduce-scatter operation followed by an all-gather operation. In this case, the reduce-scatter operation is realized by an all-to-all operation. We will denote the algorithm by the pair (*all-to-all*, *all-gather*). STAR-MPI maintains the following algorithms: (*ring, TSLR*), (*ring with light barrier, TSLR*), (*ring with MPI barrier, TSLR*), (*ring, ring*), (*ring with light barrier, right with light barrier*), and (*ring with MPI barrier, ring with MPI barrier*). We will use *Rab2-(x, y)* to denote this type of algorithms with the *(x, y)* algorithms. For example, *Rab2-(ring, ring)* denotes the *(ring, ring)* variation of this type of algorithms.

3.2 DYNAMIC AEOS ALGORITHM IN STAR-MPI

Given a set of algorithms, the primary objective of the dynamic AEOS algorithm is to find the most efficient algorithm (among the set of algorithms) for an application running on a given platform. The second objective is to minimize the overheads. Next, we will describe the AEOS algorithm and show how it achieves these two objectives.

Different MPI call sites or even the same call site invoked with different message sizes constitute different program contexts. MPI routines used in different program contexts usually have different program behavior. To achieve the maximum tuning effectiveness, routines in different contexts must be tuned independently. STAR-MPI addresses this issue as follows. For each MPI collective routine, STAR-MPI supports N independent but identical routines, where N is a parameter. Different call sites of the same MPI routine in an MPI program can be tuned independently. To deal with the case of invoking the same call site with different message sizes, STAR-MPI allows each call site to tune for a pre-defined number, X, of message sizes. If a call site has more than X different message sizes during the program execution, STAR-MPI tunes for the first X sizes and uses the default MPI routine for the rest of sizes. Note that in practice, a call site in an MPI program usually results in only a small number of message sizes, most call sites only have one message size. This arrangement allows the dynamic AEOS algorithm to focus on tuning for one message size on one call site to maximize the tuning effectiveness. In the rest of the section, we will assume that the AEOS algorithm is applied to tune one message size on one call site.

In the course of program execution, a STAR-MPI routine (for one message size in each call site) goes through two stages: *Measure_Select* and *Monitor_Adapt*. In the *Measure_Select* stage, in each invocation of the routine, one of the algorithms in the repository is used to realize the operation and the performance of the algorithm is measured. During the *Measure_Select* stage, all algorithms in the repository will be executed and measured a number of times. The number of times that each algorithm is executed and measured in this stage is a system parameter. At the end of the *Measure_Select* stage (all algorithms are executed and measured), an all-reduce operation is performed to compute the performance results on all processors and an algorithm is selected as the best algorithm based on the measured performance. The performance of all other algorithms is stored for future uses. Notice that for the whole *Measure_Select* stage, only one additional all-reduce communication (with a reasonable small message size) is performed. Note also that in this stage, less efficient algorithms end up being used to carry out the operation since their performance must be measured. After the *Measure_Select* stage, it is expected that the selected algorithm will deliver high performance for the subsequent invocations. However, this may not always occur due to various reasons. For example, the initial measurement may not be sufficiently accurate, or the workload in the application may change. To handle such situations, in the *Monitor_Adapt* stage, STAR-MPI continues monitoring the performance of the selected algorithm and adapts (changes the algorithm) when the performance of the selected algorithm deteriorates.

Figure 2 shows the details of the dynamic AEOS algorithm. In the figure, we illustrate the use of *STAR_Alltoall* to tune *MPI_Alltoall*. The AEOS algorithm is the same for all operations supported in STAR-MPI. It is important to understand that all internal states of STAR_Alltoall (or any other STAR-MPI collective operation) are static since it must be retained between invocations. Each time STAR_Alltoall is called, the algorithm first computes (line 1) the message size, x, for the operation. Once the message size is known, the algorithm can be in either of the two previously described stages depending on the value of *best_algorithm*$_x$. As shown in lines 3-6, if *best_algorithm*$_x$ points to an invalid communication algorithm index, denoted by NIL, then the algorithm is in the *Measure_Select* stage and calls the Measure_Select() routine. Otherwise, it is in the *Monitor_Adapt* stage and calls the Monitor_Adapt() routine.

The logic of the `Measure_Select()` routine (lines 7-18) is straight-forward. It runs and measures each algorithm ITER times. ITER is a parameter we control and is set to 10 in the current system. This number was determined by us experimentally; it is a trade-off between the tuning overhead and measurement accuracy. When all communication algorithms are examined, the `Dist_Time()` routine is called (line 17) to compute the communication time for all algorithms and distribute the results to all processors, and the `Sort_Alg()` routine is called (line 18) to sort the algorithms based on their performance and select the best algorithm (set the value for *best-algorithm*$_x$). Notice that the algorithm is selected based on the *best* performance measured.

Once *best_algorithm*$_x$ is set, the AEOS algorithm enters the *Monitor_Adapt* stage. In this stage, the algorithm pointed by *best-algorithm*$_x$ is used to realize the operation. The AEOS task in this stage is to monitor the performance of the selected communication algorithm and to change (adapt) to another algorithm when the performance of the selected algorithm deteriorates.

The `Monitor_Adapt()` routine is shown in lines 19-38. The logic is as follows. First, the algorithm pointed by *best-algorithm*$_x$ is used to realize the operation and the performance on each processor is measured. The monitoring is done locally (no global communication) during the monitoring period, which is defined as $\delta * ITER$ invocations, where δ is a variable whose value is initialized to be 2. At the end of the monitoring period, an all-reduce operation is performed to compute the performance of the selected algorithm and distribute the performance results to all processors. If the average communication time of the selected algorithm during the monitoring period is less than $(1 + \epsilon) * second\ best\ time$, the length of the monitoring period is doubled. Here, ϵ is a system parameter, currently set to 10%. If the average communication is more than $(1 + \epsilon) * second\ best\ time$, there are two cases. If the average communication time of the last ITER invocations is also larger than $(1 + \epsilon) * second\ best\ time$, this indicates that the selected algorithm may not be as efficient as the second best algorithm and, thus, the second best algorithm is now selected. The average time of the replaced algorithm is recorded and algorithms

```
ITER: number of iteration to examine an algorithm
δ: monitoring factor, initialized to 2
T: threshold used to switch between algorithms
TOTAL_ALGS: total number of algorithms to examine
func: pointer to a given function pointed by index_x

best_algorithm_x ← NIL;
STAR_Alltoall(sbuf, scount, stype, ...)
1   MPI_Type_size(stype, & size)
2   x ← scount * size
3   if (best_algorithm_x == NIL)
4       Measure_Select(sbuf, scount, stype..., x)
5   else
6       Monitor_Adapt(sbuf, scount, stype, ..., x)

index_x ← iter_x ← 0
Measure_Select(sbuf, scount, stype, ..., x)
7   func ← Alltoall_Alg(index_x)
8   t0 ← MPI_Wtime()
9   func(sbuf, scount, ...)
10  t1 ← MPI_Wtime()
11  le_time[index_x][iter_x] ← t1 - t0
12  iter_x++
13  if (iter_x == ITER)
14      iter_x ← 0
15      index_x++
16  if (index_x == TOTAL_ALGS)
17      Dist_Time(le_time, ge_time, best_time)
18      best-algorithm_x ← Sort_Alg(best_time, x)
Monitor_Adapt(sbuf, scount, stype, ..., x)
19  func ← Alltoall_Alg(best_algorithm_x)
20  t0 ← MPI_Wtime()
21  func(sbuf, scount, ...)
22  t1 ← MPI_Wtime()
23  total[0] ← total[0] + (t1 - t0)
24  if (δ*ITER - monitor_x ≤ ITER)
25      total[1] ← total[1] + (t1 - t0)
26  monitor_x++
27  if (monitor_x == δ * ITER)
28      MPI_Allreduce(total, ave, 2, .., MPI_SUM, ..)
29      ave[0] ← total[0] / monitor_x
30      ave[1] ← total[1] / ITER
31      if (ave[0] < (1+ε) * best_time[1])
32          δ ← δ * 2
33      else if (ave[0] ≥ (1+ε) * best_time[1])
34          if (ave[1] ≥ (1+ε) * best_time[1])
35              best_time[best_algorithm_x] ← ave[0]
36              best-algorithm_x ← Sort_Alg(best_time, x)
37              δ ← 2
38      monitor_x ← total[0] ← total[1] ← 0
Dist_Time(le_time, ge_time, best_time)
39  MPI_Allreduce(le_time, ge_time,..MPI_SUM,..)
40  foreach i in 0 .. TOTAL_ALG
41      foreach j in 0 .. ITER
42          ge_time[i][j] ← ge_time[i][j] / nprocs
43  foreach i in 0 .. TOTAL_ALG
44      best_time[i]←MIN(ge_time[i][j])
                  0≤j<ITER
```

Figure 2: Using STAR-MPI algorithm to tune MPI_Alltoall

are re-sorted based on their performance. When a new algorithm is selected, δ is reset to 2. If the average communication time of the last ITER invocations is less than $(1+\epsilon) * second\ best\ time$, the bad performance measured may be caused by some special events and the AEOS algorithm resets $\delta = 2$ so that the selected algorithm can be monitored more closely.

The monitoring is critical to ensure that STAR-MPI will eventually find an efficient algorithm. A number of trade-offs between overheads and effectiveness are made in the Monitor_Adapt routine. First, the length of the monitoring period, which is controlled by δ, doubles every time the selected algorithm continues to perform well. This reduces the monitoring overhead: if the selected algorithm continues to perform well, the total number of all-reduce operations in the *Monitor_Adapt* stage is a logarithm function of the total number of invocations. However, this creates a chance for STAR-MPI to adapt too slowly due to large monitoring periods. In practice, an upper bound can be set for δ to alleviate this problem. Second, a simple heuristic is used to decide whether the selected algorithm is still performing well. A more complex statistical approach may improve the monitoring accuracy by better filtering out noises in the measurement or program execution. Such an approach will incur more computation and more communication in the *Monitor_Adapt* stage. We adopt the simple approach since it works quite well in our experiments.

3.3 Enhancing Measure_Select by algorithm grouping

As we can see from the previous discussion, most of the tuning overheads occur in the *Measure_Select* stage. When the message size is reasonably large, the bookkeeping overheads in STAR-MPI is relatively small. However, the penalty for using less efficient algorithms to realize an operation can potentially be very high. Two parameters determine such penalty: the parameter ITER and the number of less efficient algorithms in the repository. Hence, ideally, one would like to reduce the number of algorithms as much as possible. However, the problem is that reducing the number of algorithms may make the system less robust and that before the algorithm is executed and measured, it is difficult to decide which algorithm is more efficient than other algorithms.

Algorithm grouping is one way to reduce the number of algorithms to be probed without sacrificing the tuning effectiveness. Algorithm grouping is based on the observation that a collective communication algorithm usually optimizes for one or multiple system parameters. When a system parameter has a strong impact on the performance of a collective operation, the algorithms that optimize this parameter tend to out-perform other algorithms that do not consider this parameter. Based on this observation, algorithm grouping groups algorithms based on their optimization objectives. For example, the *2D mesh*, *3D mesh*, *rdb*, and *Bruck* algorithms for *MPI_Alltoall* all try to reduce the startup overheads in the operation by reducing the number of messages. If the startup overheads in an operation is important, any of these algorithms will out-perform other algorithms that do not reduce the number of messages. Hence, these four algorithms can be joined into one group. Once all algorithms are classified into groups, the Measure_Select() routine can first identify the best performing groups by comparing algorithms in different groups (one algorithm from each group) and then determine the best performing algorithm by evaluating all algorithms in that group. This two-level tuning scheme reduces the number of algorithms to be measured in the *Measure_Select* phase while maintaining the tuning effectiveness (theoretically, all algorithms are still being considered). Notice that algorithm grouping also affects the *Monitor_Adapt* stage: in *Monitor_Adapt* stage, when a new algorithm in a new group is selected, if the algorithms in the new group have not been probed, the system must first examine all algorithms in the group before selecting the best performing algorithm.

We will call the AEOS algorithm without grouping the *basic* AEOS algorithm and with grouping the *enhanced* AEOS algorithm.

The effectiveness of algorithm grouping depends on how the algorithms are grouped. In our system, we group the algorithms based on the performance model in [3]. Algorithms that optimize the same set of parameters are classified in one group. Specifically, the 13 all-to-all algorithms are partitioned into 6 groups: group 1 contains *simple*; group 2 (used only for small messages ($\leq 256B$)) contains *rdb*, *2D mesh*, *3D mesh*, and *Bruck*; group 3 contains *ring* and *pair*; group 4 contains *ring with light barrier* and *pair with light barrier*; group 5 contains *ring with MPI barrier* and *pair with MPI barrier*; group 6 contains the two topology specific algorithms. The 12 algorithms for *MPI_Allgather* are partitioned into 6 groups: group 1 contains *simple*; group 2 contains *rdb*, *2D mesh*, *3D mesh*, and *Bruck*; group 3 contains *ring* and *pair*; group 4 contains *ring with light barrier* and *pair with light barrier*; group 5 contains *ring with MPI barrier* and *pair with MPI barrier*; group 6 contains the topology specific logical ring (TSLR) algorithm. The 20 *all-reduce* algorithms are partitioned into 3 groups based on the three types of algorithms. Notice that although this grouping scheme may not be optimal, it allows us to evaluate the algorithm grouping technique for improving dynamic AEOS scheme.

In general, grouping trades tuning overheads with the quality of the selected algorithm: the best performing algorithm may not be selected with grouping. However, in all of our tests, the enhanced STAR-MPI selected as good of (or virtually as good as) an algorithm as did the basic AEOS algorithm while significantly reducing the overheads.

4. PERFORMANCE STUDY

We perform most of the experiments on Ethernet-switched clusters since STAR-MPI is equipped with algorithms that are designed for Ethernet-switched clusters. To demonstrate the robustness of the STAR-MPI technique, we have also tested STAR-MPI on the Lemeiux machine at Pittsburgh Supercomputing Center (PSC) [19]. The nodes in our Ethernet switched clusters are Dell Dimension 2400s 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 100Mbps Ethernet switches. The topologies used in the experiments are shown in Figure 3. Part (a) of the figure is a 16-node cluster connected by a single switch while part (b) shows a 32-node cluster connected by 4 switches, with 8 nodes attached to each switch. We will refer to the topologies as topology (a) and topology (b), respectively.

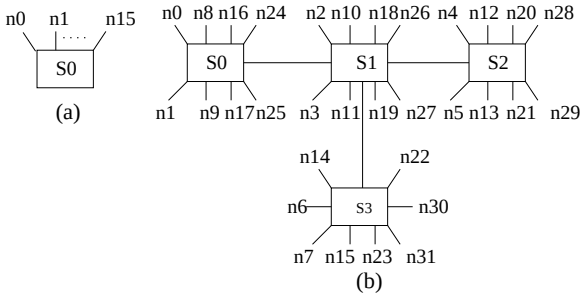


Figure 3. Topologies used in the experiments.

As discussed earlier, there are two major performance issues in STAR-MPI. First, for STAR-MPI to be efficient, the AEOS technique must be able to select good communication algorithms at runtime. To examine the capability of STAR-MPI in selecting good communication algorithms, we compare the performance of STAR-MPI (STAR) with the original MPICH 2.1.0.1 and our static tuning system [3] (STATIC), which also runs on MPICH and tunes algorithms for Mpptest [7]. The algorithm repository in STAR-MPI is a subset of the algorithm repository in STATIC. Second, STAR-MPI introduces overheads in both the `Measure_Select` and `Monitor_Adapt` stages. In the `Measure_Select` stage, less efficient communication algorithms are executed to carry out the operations. We denote such overheads as O^c_{MS}. Additional overheads are introduced in this stage to execute the AEOS logic (e.g. measuring, computing, and recording the performance of all communication algorithms). We denote such overheads as O^a_{MS}. In the `Monitor_Adapt` stage, the overheads are introduced to monitor the performance and to execute the logic to determine whether the current algorithms should be changed. We denote the overheads in the `Monitor_Adapt` stage as O_{MA}. In the rest of the section, we will first compare the basic AEOS algorithm with the enhanced AEOS algorithm that uses algorithm grouping. We will then study the capability of STAR-MPI in selecting good communication algorithms in a number of application programs. After that, we evaluate the overheads of STAR-MPI. Finally, we present our experiments on the Lemieux cluster at PSC.

4.1 Basic AEOS .vs. Enhanced AEOS

For STAR-MPI to be efficient, it must (1) be able to find the efficient algorithms and (2) find the algorithms as quickly as possible. The algorithm found at the end of the *Measure_Select* stage usually (but not always) offers reasonably good performance and is used thereafter. Hence, a good indication of the performance of an AEOS algorithm is the (1) the quality of the algorithm selected at the end of *Measure_Select*, and (2) the duration of the *Measure_Select* stage, which measures how fast the AEOS algorithm can find the selected algorithm. This section compares the basic and enhanced AEOS algorithms with these two metrics.

```
for (i = 0; i < 500; i++) {
    ... // computation that lasts roughly 5 times
        // the collective operation time
    start = MPI_Wtime();
    MPI_Alltoall(...);
    elapsed += (MPI_Wtime() - start);
}
```

Figure 4. An example micro-benchmark.

We use micro-benchmarks that are similar to the one shown in Figure 4 in the comparison. This micro-benchmark simulates programs with a perfect computation load distribution. The main loop contains both computation and collective communication. The time for the computation in the loop is set to be roughly 5 times the total communication time. The elapsed time for the communication is measured and reported.

Table 1 shows the number of invocations in the *Measure_Select* stage, the total communication time in this stage, and the algorithms selected by the basic and enhanced AEOS algorithms. As expected, the enhanced scheme greatly reduces the number of invocations and the time in the *Measure_Select* stage. Moreover, the algorithms selected by the two schemes are mostly the same. In cases when the selected algorithms are different (e.g. 128KB all-to-all and 128KB all-reduce), the performance of the different communication algorithms is very similar. We have conducted experiments with different message sizes and different topologies and obtained similar observations. Hence, we conclude that the enhanced scheme is more efficient than the basic scheme (at least in our system). In

Table 1. Basic AEOS vs. enhanced AEOS on topology (a).

operation	msg size	Measure_ Select	basic	enhanced
all-to-all	2KB	# of invocations	90	50
		time (ms)	2181	1183
		algorithm	*simple*	*simple*
	128KB	# of invocations	90	60
		time (ms)	31953	19018
		algorithm	*ring light*	*pair light*
all-gather	2KB	# of invocations	120	60
		time (ms)	2055	1229
		algorithm	*simple*	*simple*
	128KB	# of invocations	120	60
		time (ms)	41233	18335
		algorithm	*TSLR*	*TSLR*
all-reduce	2KB	# of invocations	200	110
		time (ms)	2413	1571
		algorithm	*Rab1-2D*	*Rab1-2D*
	128KB	# of invocations	200	110
		time (ms)	25319	7895
		algorithm	*Rab1-3D*	*Rab1-Bruck*

the rest of the section, we will only report results of STAR-MPI with the enhanced AEOS algorithm.

4.2 Application Results

Since STAR-MPI targets programs that run for a large number of iterations, we select the application benchmarks that (1) run for a large number of iterations and (2) have significant collective communications. To achieve high performance for this type of programs, it is critical that STAR-MPI must eventually select efficient communication algorithms to carry out the collective operations. The results in the this sub-section mainly reflect the capability of STAR-MPI in selecting communication algorithms.

We use four applications in the evaluation: FFTW [8], LAMMPS [13], NTUBE [22], and NBODY [17]. FFTW [8] is a C library of routines for computing the discrete Fourier transform in one or more dimensions, of arbitrary input size, and of both real and complex data. When using the benchmark test driver, the value of l (linear size) is 1500 and the value of *nfft* (number of Fourier transforms to execute) is 500. The LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) [13] benchmark models the assembly of particles in a liquid, solid, or gaseous state. In the experiments, we ran the program with 1720 copper atoms for 10000 iterations. The NTUBE (Nanotube) program performs molecular dynamics calculations of thermal properties of diamond [22]. In the evaluation, NTUBE runs for 1000 steps and simulate 25600 atoms. Finally, NBODY [17] simulates over time steps the interaction, in terms of movements, position and other attributes, among the bodies as a result of the net gravitational forces exerted on one another. We ran the code for 1000 steps with 8000 bodies on topology (a) and 25600 bodies on topology (b). Note that the number of iterations or time steps for each benchmark is chosen such that it is sufficient enough (1) to allow STAR-MPI routines finish the *Measure_Select* stage and (2) to achieve considerable performance gains that will amortize the overheads associated with the STAR-MPI technique. For the different benchmarks, Table 2 shows the major MPI collective routines and the message sizes for topologies (a) and (b). These routines account for a significant portion of the total application times and are tuned using STAR-MPI.

Table 2. Collective operations in the applications on different topologies (T).

program	operation	msg size
FFTW	MPI_Alltoall	T.(a): 141376B T.(b): 33856B
LAMMPS	MPI_Allreduce	T.(a): 42382B T.(b): 42382B
NTUBE	MPI_Allgatherv	T.(a): 256000B T.(b): 128000B
NBODY	MPI_Allgather	T.(a): 20000B T.(b): 32000B

Next, Table 3 shows the different communication algorithms selected in STAR, STATIC, and MPICH to realize the collective operations in the four application benchmarks. The STAR algorithms in the table are the final algorithms selected for the application. Several algorithms in the table are not included in STAR-MPI and thus have not been discussed. The description of these algorithms can be found in [3]. There are two main observations. First, MPICH has limited software adaptability as it only considers the message size and the number of processors. In particular, for all benchmarks except NBODY, the communication algorithms that MPICH uses are the same across both topologies. In the case of NBODY, the message sizes of the all-gather operation on the two topologies were not in the same range, which caused MPICH to use two different communication algorithms. Since MPICH does not take into considerations application behavior and all aspects of the architecture (or platform), its predetermined selection of algorithms will be inadequate in many cases. Second, with the exception of NTUBE and NBODY on topology (b), the table shows that the STAR-MPI versions of communication algorithms for the different collective operations are quite different than (and superior to) the ones used by STATIC. The reason these algorithms are picked by STAR-MPI but not STATIC, although the algorithms are available in its repository, is that STATIC has only architectural information and selects the best algorithms for Mpptest, not the applications. As a result, STATIC may not yield the most efficient algorithm for an application since the program context is unavailable. STAR-MPI attains full adaptability for the collective operations because it has access to application information.

The results for the application benchmarks for MPICH, STAR, and STATIC are summarized in Table 4. Note that in all cases, STAR overhead is included in the results presented, while MPICH and STATIC both have no run-time overhead. First, the ability of STAR to select better communication algorithms than MPICH and STATIC is evident in the significant performance gains shown in the table. For all benchmarks running on the two topologies, except for NTUBE on topology (a), STAR is superior to MPICH. For example, for the FFTW benchmark, STAR achieves a 64.9% and 31.7% speed ups over MPICH on topology (a) and topology (b), respectively. Also, substantial gains are seen for LAMMPS (85.6%) and NTUBE (408.9%) on topology (b). The result for the NTUBE benchmark on topology (a) shows that STAR performs slightly worse than MPICH. This is because both STAR and MPICH use the same communication algorithm to realize the all-gatherv operation in the benchmark, with STAR paying the extra tuning overhead.

Comparing STAR to STATIC shows that in many cases STAR is also superior. For example, on FFTW, STAR speedup relative to MPICH is much larger than that of STATIC (65% to 11% on topology (a) and 32% to 7.7% on topology (b)). STAR speedup is also much greater on LAMMPS (b) and NBODY (a), and STAR

Table 3. Communication algorithms used in STAR, STATIC, and MPICH on different topologies (T).

	T.	STAR	STATIC	MPICH
FFTW	(a)	*pair with light barrier*	*pair with MPI barrier*	*pair*
(*MPI_Alltoall*)	(b)	*pair with light barrier*	*tuned pair with N MPI barrier*[3]	*pair*
LAMMPS	(a)	*Rab1-ring with light barrier*	*Rab1-tuned*[3]	*MPICH Rab.* [3]
(*MPI_Allreduce*)	(b)	*Rab1-rdb*	*Rab2-(tuned, tuned)* [3]	*MPICH Rab.*
NTUBE	(a)	*TSLR*	*pair with MPI barrier*	*logical ring* [3]
(*MPI_Allgatherv*)	(b)	*TSLR*	*TSLR*	*logical ring*
NBODY	(a)	*simple*	*TSLR*	*rdb*
(*MPI_Allgather*)	(b)	*TSLR*	*TSLR*	*logical ring*

does not slow down on NTUBE (a), as described earlier, whereas STATIC does. This demonstrates the effectiveness of STAR that has a subset of algorithms in selecting better communication algorithms than STATIC, which has a super-set of algorithms. In two of the other cases (LAMMPS (a), NBODY (b)), the performance is similar, with STATIC slightly better. The one exception is on NTUBE (b), where STATIC speedup is much larger than STAR. We look at these last three cases next.

Table 4. Application completion times (seconds) on different topologies (T).

program	T.	STAR	STATIC	MPICH
FFTW	(a)	350.8	519.6	578.6
	(b)	636.3	778.0	838.1
LAMMPS	(a)	9780	9515	11040
	(b)	1991	2432	3696
NTUBE	(a)	568.0	725.0	566.0
	(b)	758.4	601.0	3860
NBODY	(a)	4002	4268	4304
	(b)	2167	2120	2946

Table 5 shows the performance of STAR with and without overhead, relative to STATIC. The performance of STAR without overhead, denoted as STAR', is obtained by running the final routine selected by STAR-MPI without the tuning and monitoring overheads. From Figure 5, we can see that STAR-MPI without overheads performs at least as good as STATIC, which indicates that the performance penalty (versus STATIC) is due to the overheads. As will be shown in the next sub-section, the overhead is mainly introduced in the *Measure_Select* stage. The AEOS algorithm in STAR-MPI is robust. If applications run for more iterations, the tuning overheads will be amortized. For example, if NTUBE runs for 2000 instead of 1000 time steps, the absolute STAR overhead would remain roughly the same, while the relative STAR overhead would decrease substantially. Notice that STATIC also has overhead: STATIC must be run over a significant period of time when a new platform is encountered. We are not considering STATIC overheads in this experiment. Note also that the NTUBE (a) result shows that tuning with Mpptest can sometimes lead to the algorithms that significantly degrade the performance for an application. This is a major limitation of STATIC.

Table 5. Application completion times (seconds).

program	Topo.	STAR	STATIC	STAR'
LAMMPS	(a)	9780	9515	9420
NTUBE	(b)	758.4	601.0	601.0
NBODY	(b)	2167	2120	2120

4.3 STAR-MPI overhead

Using the micro-benchmarks similar to the code in Figure 4, we examine in depth the overhead of STAR-MPI. We measure the overhead introduced by the execution of less efficient algorithms, O^c_{MS}, and the overheads for running the AEOS algorithm in both stages of STAR-MPI, namely O^a_{MS} and O_{MA}. Note that besides the O_{MA} overhead in the *Monitor_Adapt* stage, STAR-MPI may introduce extra overheads in this stage if it adapts to a different algorithm. While we occasionally observe such adaptation (all such adaption occurs in the first monitoring period in our experiments), it is a low probability random event. Hence, we only evaluate O^a_{MS}, O^c_{MS}, and O_{MA}. In the following, we first look at the per invocation time of STAR-MPI collective routines in the *Measure_Select* and *Monitor_Adapt* stages, and then break down the time in terms of the different overheads.

Table 6. Per invocation time (ms) for collective operations in the micro-benchmark.

operation	T.	msg size	MPICH	STAR	
				Measure_ Select	Monitor_ Adapt
all-to-all	(a)	16KB	32.0	46.4	27.0
		64KB	305.7	192.1	114.6
		256KB	519.4	658.7	498.6
	(b)	16KB	324.0	366.4	323.2
		64KB	1493	1366	1108
		256KB	6079	7154	5716
all-gather	(a)	16KB	31.8	46.0	25.5
		64KB	111.6	147.2	104.5
		256KB	446.0	596.8	416.9
	(b)	16KB	87.8	293.8	58.66
		64KB	1532	1232	542
		256KB	6432	5037	2004
all-reduce	(a)	16KB	5.9	12.0	4.6
		64KB	18.7	24.4	18.6
		256KB	68.3	95.5	68.4
	(b)	16KB	18.4	26.4	9.68
		64KB	82.0	76.8	34.3
		256KB	335.4	250	128.6

The per invocation times in the *Measure_Select* and *Monitor_Adapt* stages of STAR-MPI all-to-all, all-gather, all-reduce routines with different message sizes on topology (a) and (b) are shown in Table 6. The results are obtained using the micro-benchmark with 500 iterations, which include the iterations for both the *Measure_Select* and *Monitor_Adapt* stages. For example, for all-gather on topology (b) with message size 64KB, the *Measure_Select* stage occupies 60 invocations and the *Monitor_Adapt* stage occupies 440 invocations. For reference, the per invocation time for MPICH is also shown. There are a number of common observations for all

operations on both topologies. First, the per invocation times are very different for the *Measure_Select* stage and the *Monitor_Adapt* stage. This is because the best performing algorithm significantly out-performs some of the algorithms in the repository. Second, as shown in the table, although the per invocation time of STAR-MPI in the *Measure_Select* stage reflects a quite significant overhead, such overhead is amortized (and then offset) by the gains due to a better communication algorithm during the post tuning or *Monitor_Adapt* stage. Third, in some cases (e.g. all-gather on topology (b) with message sizes of 64KB and 256KB), STAR-MPI outperforms MPICH even in the *Measure_Select* stage. This is because some of the communication algorithms that STAR utilizes during tuning are more efficient than MPICH.

Table 7. Per invocation overheads (in millisecond) for collective operations in the micro-benchmark.

operation	T.	msg size	O^a_{MS}	O^c_{MS}	O_{MA}
all-to-all	(a)	16KB	0.04	46.4	0.01
		64KB	0.35	191.8	0.06
		256KB	1.60	657.1	0.30
	(b)	16KB	0.01	366.4	0.4
		64KB	0.01	1366.0	1.4
		256KB	0.01	7153.9	5.8
all-gather	(a)	16KB	0.01	45.9	0.03
		64KB	0.01	147.2	0.1
		256KB	0.7	596.1	0.2
	(b)	16KB	0.01	293.8	0.08
		64KB	0.01	1232	0.8
		256KB	0.01	5037.0	0.6
all-reduce	(a)	16KB	0.02	12.0	0.02
		64KB	0.01	24.39	0.03
		256KB	0.01	95.4	0.05
	(b)	16KB	0.05	26.3	0.03
		64KB	0.15	76.5	0.07
		256KB	0.5	249.5	0.20

Table 7 breaks down the per invocation time in terms of the O^a_{MS}, O^c_{MS}, and O_{MA} overheads for the same STAR-MPI collective routines. For the different message sizes, the table shows that O^a_{MS} and O_{MA} are very small and account for less than 0.3% of the per invocation times, shown previously in Table 6, for the *Measure_Select* stage or the *Monitor_Adapt* stage. On the other hand, we observe that O^c_{MS} can be very large. Thus, most of the overhead of STAR-MPI is due to the communication overhead in the tuning phase. This indicates that the selection of the set of communication algorithms is very critical for STAR-MPI to achieve high performance. Moreover, for different topologies, the table shows that STAR-MPI may introduce very different overheads. For example, the STAR-MPI all-gather routine introduces much more overhead on topology (b) than that on topology (a). This is because the topology can significantly affect the performance of a collective communication algorithm. Since the impact of topology is so significant, it may be worthwhile to develop a performance model that can take network topology into account and use the prediction from such a model to reduce the number of algorithms to be probed.

4.4 STAR-MPI on Lemieux

To further study the effectiveness and impact of our STAR-MPI technique on different platforms, we perform experiments on Lemieux, a supercomputing cluster located in Pittsburgh Supercomputing Center (PSC) [19]. The machine consists of 750 Compaq Alphaserver ES45 nodes, each of which includes four 1-GHz SMP processors with 4GB of memory. The nodes are connected with a Quadrics interconnection network, and they run Tru64 Unix operating system. The experiments are conducted with a batch partition of 128 processors running on 32 dedicated nodes although other jobs were concurrently using the network. The benchmarks are compiled with the native *mpicc* on the system and linked with the native MPI and ELAN libraries. ELAN is a low-level internode communication library that efficiently realizes many features of the Quadrics interconnection such as multicast. We will use NATIVE to denote the performance of the native MPI routines.

The algorithms used in the collective communication routines in the native MPI library are unknown to us. The Quadrics interconnect in this machine has very efficient hardware support for multicast. As a result, for collective operations that have a multicast or broadcast component, including all-gather, all-gatherv, and all-reduce, the native routines out-perform STAR-MPI (sometimes to a large degree) since all STAR-MPI algorithms are based on point-to-point primitives. However, STAR-MPI all-to-all routine offers better performance than the native routine on this cluster.

The micro-benchmark results for MPI_Alltoall (500 iterations) on Lemieux are shown in Figure 5. Part (a) of the figure shows the results for small messages while part (b) shows the results for medium/large messages. As shown in both parts of the figure, for all message sizes, STAR offers higher performance than NATIVE, especially as the message size increases. For example, when the message size is 256KB, the respective completion times for STAR and NATIVE are 1005.6ms and 1329.5ms with STAR achieving a speed up of 32.3%. Similar performance trend is evident when experimenting on the FFTW application benchmark with $l = 5700$ and $nfft = 500$. The execution time for the benchmark using STAR-MPI is 184.3s as opposed to 212.8s for NATIVE, which is a 15.5% speed up.

Although the communication algorithms in STAR-MPI were designed for Ethernet-switched clusters, not for Lemieux, as shown in this experiment, STAR-MPI can improve performance for other types of clusters. This demonstrates the robustness of the proposed delayed finalization technique.

5. CONCLUSION

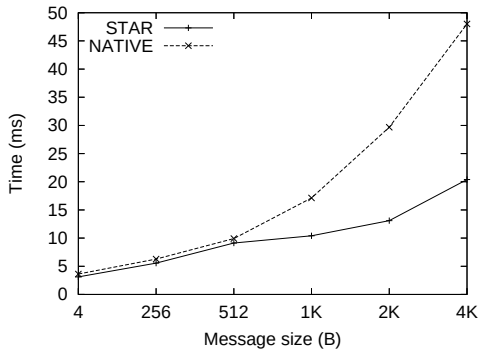
In this paper, we proposed the delayed finalization of MPI collective communication routines (DF) technique to improve the software adaptability of MPI libraries. Using the DF approach, we developed STAR-MPI. STAR-MPI is able to achieve adequate software adaptability for MPI collective routines, which many library implementations of MPI fall short of. By utilizing a set of different communication algorithms that perform well for different situations and then empirically measuring the performance of these algorithms in the context of both the application and platform, STAR-MPI yields efficient, customized MPI collective routines for the application on the platform. Performance results showed that STAR-MPI routines incur reasonable overheads, deliver high performance, and significantly out-perform the routines in the traditional MPI library in many cases.

References

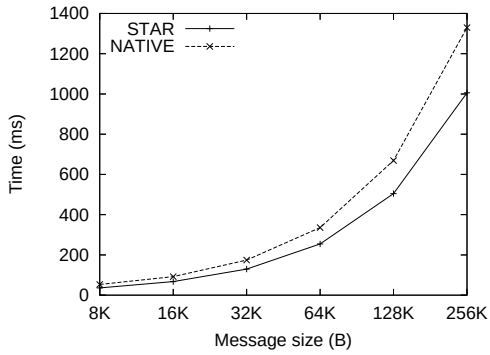
[1] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. "Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology." In *Proceedings of the ACM SIGARC International Conference on SuperComputing*, 1997.

[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. "Automatic Generation and Tuning of MPI Collective Communication Routines." *The 19th ACM International Conference on Supercomputing (ICS)*, Cambridge, Massachusetts, June 20-22, 2005.



(a) small messages



(b) medium-large messages

Figure 5. All-to-all micro-benchmark results on 128 processors (Lemieux), average per invocation time.

[4] A. Faraj and X. Yuan. Message Scheduling for All–to–all Personalized Communication on Ethernet Switched Clusters. *IEEE IPDPS*, April 2005.

[5] A. Faraj and X. Yuan. "Bandwidth Efficient All-to-All Broadcast on Switched Clusters." *The 2005 IEEE International Conference on Cluster Computing*, Boston, MA, Sept 27-30, 2005.

[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] William Gropp and Ewing Lusk, "Reproducible Measurements of MPI Performance Characteristics." *Technical Report ANL/MCS-P755-0699*, Argonne National Labratory, Argonne, IL, June 1999.

[8] FFTW. http://www.fftw.org.

[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] 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.

[12] T. Kielmann, et. al. Magpie: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *ACM SIGPLAN PPoPP*, pages 131–140, May 1999.

[13] LAMMPS: Molecular Dynamics Simulator, Available at http://www.cs.sandia.gov/ sjplimp/lammps.html.

[14] 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.

[15] MPICH - A Portable Implementation of MPI. Available at http://www.mcs.anl.gov/mpi/mpich.

[16] H. Ogawa and S. Matsuoka. OMPI: Optimizing MPI Programs Using Partial Evaluation. In *Supercomputing'96*, November 1996.

[17] Parallel N-Body Simulations, Available at http://www.cs.cmu.edu/ scandal/alg/nbody.html.

[18] P. Patarasuk, A. Faraj, and X. Yuan. "Pipelined Broadcast on Ethernet Switched Clusters." *The 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, Rhodes Island, Greece, April 25-29, 2006.

[19] Pittsburg Supercomputing Center, Available at http://www.psc.edu/machines/tcs/lemieux.html.

[20] R. Rabenseifner, "A new optimized MPI reduce and allreduce algorithms." Available at http://www.hlrs.de/organization/par/services/models /mpi/myreduce.html, 1997.

[21] R. Rabenseinfner, "Automatic MPI counter profiling of all users: First results on CRAY T3E900-512," In *Proceedings of the Message Passing Interface Developer's and User's Conference*, pages 77-85, 1999.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically Tuned Collective Communications. In *Proceedings of SC'00: High Performance Networking and Computing*, 2000.

[27] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing'98: High Performance Networking and Computing*, 1998.