An Empirical Approach for Efficient All-to-All Personalized Communication on Ethernet Switched Clusters [*]

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

Abstract

All–to–all personalized communication (AAPC) is one of the most commonly used communication patterns in parallel applications. Developing an efficient AAPC routine is difficult since many system parameters can affect the performance of an AAPC algorithm. In this paper, we investigate an empirical approach for automatically generating efficient AAPC routines for Ethernet switched clusters. This approach applies when the application execution environment is decided, and it allows efficient customized AAPC routines to be created. Experimental results show that the empirical approach generates routines that consistently achieve high performance on clusters with different network topologies. In many cases, the automatically generated routines out-perform conventional AAPC implementations to a large degree.

Keywords: All–to–all personalized communication, Ethernet switched cluster, empirical technique, cluster computing, MPI.
Technical areas: MPI, cluster computing.

1. Introduction

All–to–all personalized communication (AAPC) is one of the most common communication patterns in high performance computing. In AAPC, each node in a system sends a different message of the same size to every other node. The Message Passing Interface (MPI) routine that realizes AAPC is *MPI_Alltoall* [14]. AAPC appears in many high performance applications, including matrix transpose, multi-dimensional convolution, and data redistribution.

Switched Ethernet is the most widely used local–area–network (LAN) technology. Many Ethernet–switched clusters of workstations are used to perform high performance computing. For such clusters to be effective, communications must be carried out as efficiently as possible.

Although a large number of AAPC algorithms have been proposed [1, 5, 8, 9, 12, 16, 18, 21], developing an efficient AAPC routine is still a challenging task. The main challenge comes from the fact that many system parameters can significantly affect the performance of an AAPC algorithm. These system parameters include operating system context switching overheads, the ratio between the network and the processor speeds, the switch design, the amount of buffer memory in switches, and the network topology. These parameters are difficult to model, and it is virtually impossible for the routine developer to make the right choices for different platforms.

The AAPC routines in existing communication libraries such as MPICH [15, 19] and LAM/MPI [11] are implemented before the application execution environment is decided. As a result, these routines do not sufficiently incorporate the knowledge of the system parameters. For example, topology specific algorithms cannot be used since the routines are implemented before the topology is decided. Hence, although these routines correctly carry out the AAPC operation on different platforms, they cannot achieve high performance in many cases.

In this paper, we investigate an empirical approach for automatically generating efficient AAPC routines for Ethernet switched clusters. This approach applies when the application execution environment is decided. Since the AAPC routines are finalized after the execution environment is fixed, architecture specific algorithms can be considered. Furthermore, while it is difficult to model system parameters, the overall impact of the system parameters on AAPC performance can be measured empirically. By utilizing the platform specific information and employing the empirical approach to select the best implementations, it is possible to create an efficient customized AAPC routine for a partic-

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

ular platform.

We implemented an empirical based AAPC routine generator for Ethernet switched clusters. The generator includes a module that takes topology information as input and produces topology specific AAPC implementations. In addition, the generator also maintains an extensive set of topology independent AAPC implementations that can potentially achieve high performance in different situations. The implementations in the final customized AAPC routine are selected from the topology independent and topology specific implementations using an empirical approach.

We present AAPC algorithms maintained in the generator and describe the method to generate the customized AAPC routines. We carried out extensive experiments on Ethernet switched clusters with many different network topologies. The results indicate that the generated routines consistently achieve high performance on clusters with different network topologies. In many cases, the generated routines out-perform conventional AAPC implementations, including the ones in LAM/MPI [11] and a recently improved MPICH [19], to a large degree.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 discusses the network and performance models. Section 4 presents AAPC algorithms. Section 5 describes the procedure to generate the customized routines. Section 6 reports experimental results. Finally, the conclusions are presented in Section 7.

2. Related Work

AAPC has been extensively studied due to its importance. A large number of optimal AAPC algorithms for different network topologies with different network models were developed. Many of the algorithms were designed for specific network topologies that are used in parallel machines, including hypercube [9, 21], mesh [1, 8, 16, 18], torus [12], k-ary n-cube [21], and fat tree [4]. Topologies independent AAPC algorithms were proposed in [8, 19]. Heuristic algorithms were developed for AAPC on irregular topologies [13]. A framework for AAPC that is realized with indirect communications was reported in [10]. Efficient AAPC scheduling schemes for clusters connected by a single switch was proposed in [17]. While the existing AAPC algorithms take some aspects of the architecture into consideration, they may not achieve the best performance in practice since other system parameters may also affect the performance. This paper is not concerned with a particular AAPC algorithm, instead, it focuses on using an empirical approach to obtain customized AAPC routines that are efficient in practice. The empirical technique has been applied successfully to various computational library routines [6, 22]. This approach has also been applied to tune one-to-all and one-to-many types of collective communications [20]. The issues in one-to-all and one-to-many communications are different from those in AAPC.

3. Performance model

We will describe the cost model that we use to estimate the communication performance of the algorithms. Our model reflects the following costs:

1. *Per pair communication time.* The time taken to send a message of size n between any two nodes can be modeled as $\alpha + n\beta$, where α is the startup overhead and β is the per byte transmission time.

2. *Sequentialization overheads.* Some algorithms partition AAPC 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 may limit the parallelism in AAPC. We use the notation θ to denote the sequentialization overheads between 2 phases. For an algorithm that realizes AAPC in m phases, the sequentialization overheads is $(m-1)\theta$.

3. *Synchronization overheads.* Many AAPC algorithms introduce synchronization to reduce network contention. There are two types of synchronizations, the system wide synchronization and the light-weight synchronization that ensures that a communication happens before another. The light-weight synchronization, also called *light barrier*, is usually implemented by having one node send a zero byte message to another node. We will use δ_l to denote the cost of a light-weight synchronization. The system wide synchronization is typically implemented with a barrier operation (e.g. *MPI_Barrier* in MPI). We will use δ_h to denote the cost of a system wide synchronization. In most cases, δ_h is larger than δ_l, which in turn, is larger than θ.

4. *Contention overheads.* Contention can happen in three cases: *node contention* when more than one node tries to send messages to the same node, *link contention* when more than one communication uses the same links in the network, and *switch contention* when the amount of data passing a switch is more than what the switch can handle. We will use γ_n to denote node contention, γ_l for link contention, and γ_s for switch contention. 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 communication is expressed in five terms: the startup time term that is a multiple of α, the bandwidth term that is a multiple of $n\beta$, the sequentialization costs term that is a multiple of θ, the synchronization costs term that is a multiple of δ_l or δ_h, and the contention costs term that is a combination of γ_n, γ_l, and γ_s. Notice that some parameters in the model such as the sequentialization costs and contention costs can contribute significantly to the overall communication costs; however, they are difficult to quantify. In practice, they cannot be measured accurately since they are non-deterministic in nature. As a result, this cost model is used only to justify the selection of algorithms, but not to accurately predict the performance.

4. AAPC algorithms

This section describes the potential AAPC algorithms, including both topology independent algorithms and topology specific algorithms. We will assume that p is the number of processes and n is the message size.

4.1. Algorithms for small messages

In general, algorithms for AAPC with small messages are concerned about reducing the startup and sequentialization costs while algorithms for large messages usually attempt to reduce the contention and the synchronization costs. Since the startup and sequentialization costs do not depend on the network topology, all algorithms for small messages are topology independent.

Simple. 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 $i \rightarrow 0$, $i \rightarrow 1$, ..., $i \rightarrow p-1$. The estimated time is $(p-1)\alpha + (p-1)n\beta + \gamma$. This algorithm does not have sequentialization overheads since all communications are carried out in one phase.

Spreading Simple. This algorithm is similar to the simple algorithm except that the order of communications for node i is $i \rightarrow i+1$, $i \rightarrow i+2$, ..., $i \rightarrow (i+p-1)\ mod\ p$. By changing the order of communications, node contention may potentially be reduced. The estimated time is the same as that for the simple algorithm except that the γ term might be smaller.

Simple and spreading simple algorithms minimize the sequentialization costs. Another extreme is to minimize the startup costs. The **Bruck**[2] and **recursive doubling** algorithms achieve this by minimizing the number of messages that each node sends at the cost of sending extra data.

Recursive doubling. This algorithm first performs an all-gather operation, which distributes all data in each node to all other nodes, and then copies the right portion of the data to the receiving buffer. Thus, in terms of communication, the recursive doubling algorithm for AAPC is similar to the recursive doubling algorithm for an all-gather operation with a message size of pn. Details about this algorithm can be found in [19]. 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 [19].

Bruck. This is another $lg(p)$-step algorithm that sends a less amount of extra data in comparison to the recursive doubling algorithm. Details can be found in [2, 19]. When the number of processes is a power of two, the estimated time is $lg(p)\alpha + \frac{np}{2}lg(p)\beta + (lg(p)-1)\theta + \gamma$. The startup costs term and the sequentialization costs terms for this algorithm are exactly the same as those for the recursive doubling algorithm. However, the bandwidth term is smaller. This algorithm also works with slightly larger overheads when the number of processes is not a power of two.

Between the two extremes, the generator maintains the **2D mesh** and **3D mesh** algorithms that represent compromises between minimizing the number of messages and minimizing the number of phases. Like the recursive doubling algorithm, both algorithms perform all-gather followed by a copy.

2D mesh. This algorithm applies only when $p = x \times y$ where the processes are organized as a logical $x \times y$ mesh. The algorithm tries to find the factoring such that x and y are close to $\sqrt{p}$. In the worst case, $x = 1$, $y = p$, and this algorithm degenerates to the simple algorithm. Assume that $x = y = \sqrt{p}$, the all-gather operation is first performed in the x dimension with message size equals to pn and then in the y dimension with message size equal to $p\sqrt{p}n$. Thus, the estimated time is $2(\sqrt{p}-1)\alpha + (p-1)pn\beta + \theta + \gamma$. Compared to the simple algorithm, this algorithm sends a smaller number of messages, but more data. The communications in this algorithm are carried out in two phases resulting in a θ term in the estimated time. Compared to the recursive doubling algorithm, this algorithm sends a larger number of messages in a smaller number of phases.

3D mesh. This algorithm applies only when $p = x \times y \times z$ where the processes are organized as a logical $x \times y \times z$ mesh. The algorithm tries to find the factoring such that x, y, and z are close to $\sqrt[3]{p}$. In the worst case, it degenerates to the 2D mesh algorithm or even the simple algorithm. 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 the 2D mesh algorithm, this algorithm sends a smaller number of messages, but consists of three phases, which introduce a 2θ sequentialization overhead.

4.2. Algorithms for large messages

To achieve high performance for AAPC with large messages, one of the most important issues is to reduce network contention in the system. However, it must be noted that a contention free AAPC algorithm may not achieve the best performance in practice since most systems can handle a certain degree of contention effectively, which indicates that algorithms that allow a limited degree of contention will likely offer the highest performance. Next, we will describe the AAPC algorithms for large messages.

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, there is no node contention if all phases execute in a lock-step fashion. Since different nodes may finish a phase and start a new phase at different times, this algorithm only reduces node contention. The ring algorithm does not consider switch contention and link contention. The estimated time is $(p-1)(\alpha + n\beta) + (p-2)\theta + \gamma_n + \gamma_s + \gamma_l$.

Ring with light barrier. In this algorithm, light barriers are added between the communications in different phases to eliminate potential node contention. Note that we assume that node contention happens when more than one node sends large data messages to a receiver and ignore the node contention caused by a large data message and a tiny synchronization message. 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 barriers while reducing the contention overheads.

Ring with MPI barrier. The ring with light barrier algorithm allows the phases to proceed in an asynchronous manner which may cause too many data to be injected into the network. In this algorithm, MPI barriers are added between two phases, which makes the phases execute in a lock-step fashion resulting in a less likely switch contention since the total amount of data in the network at a given time is less than the amount of data transferred in one phase. 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 increases synchronization overheads while reducing the contention overheads.

Ring with N MPI barriers. 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 N, $1 \le N \le p-2$, barriers in the whole communication. An MPI barrier is added every $\frac{p-1}{N+1}$ 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 family of algorithms contains $p-2$ different algorithms (the potential value for N being $1..p-2$): **Ring with $N = 1$ MPI barriers**, **Ring with $N = 2$ MPI barriers**, ..., **Ring with $N = p-2$ MPI barriers** algorithms. These algorithms are implemented as one routine with an algorithm parameter N.

Pair. This 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 (phases). In step i, node j exchanges a message with 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 independent 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 [5]. This algorithm finds the optimal message scheduling for a system with any number of Ethernet switches. The idea is to partition 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. To prevent communications in different phases from affecting each other, light weight barriers are added. Details about this algorithm can be found in [5]. The estimated time for this algorithm depends on the topology.

5. Generating AAPC routines

Armed with the algorithms described in the previous section, the automatic AAPC routine generator uses an empirical approach to determine the best AAPC implementations for a particular platform and for different ranges of message sizes. The generator produces a customized AAPC routine in three steps.

1. The topology specific implementation is generated using the topology information. After this step, the topology specific routine is treated the same as other topology independent routines.
2. The best implementations are determined for a set of message sizes, which includes 1B, 64B, 256B, 1KB, 2KB, 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, and 256KB. For each message size, the best implementation is selected by running all algorithms for the size and empirically measuring the performance. The performance measurement follows the approach in Mpptest [7], which is shown in Figure 1.

```
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();
for (count = 0; count < ITER_NUM; count ++) {
  alltoall_simple(...);
}
elapsed_time = MPI_Wtime() - start;
```

Figure 1. Measuring AAPC performance.

3. The exact message sizes that warrant the changes of algorithms are determined in this step. The generator logically partitions message sizes into ranges including $(1B, 64B)$, $(64B, 256B)$, ..., $(128KB, 256KB)$. The system assumes the same algorithm for message sizes $\geq$ 256KB. The system examines each range (s, e). If the same implementation is associated with both s and e, then the implementation is selected for the whole range (s, e). If s and e are associated with different implementations, let I_s be the implementation associated with s and I_e be the implementation associated with e. The generator will determine the size $msize$, $s \leq msize \leq e$, that the algorithm should change from I_s to I_e, that is, I_s is selected for range $(s, msize)$ and I_e is selected for range $(msize, e)$. A binary search algorithm is used to determine $msize$. For each point in the binary search, the performance of both I_s and I_e is measured and compared. Once implementations are decided for the different ranges of message sizes, the generator uses this information to produce the final customized routine.

6. Experiments

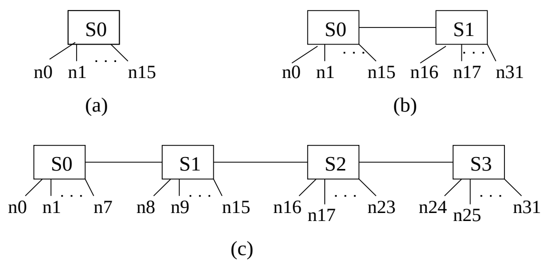


Figure 2. Topologies used in the experiments

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.

We conducted experiments on different topologies that are depicted in Figure 2. Figure 2 (a) is a 16-node cluster connected by a single switch. Clusters connected by a single Ethernet switch is common in practice. Figure 2 (b) is a 32-node cluster, connected by 2 switches with 16 nodes on each switch. Part (c) is a 32-node cluster of 4 switches with 8 nodes on each switch. We will refer to these topologies as *topology (a)*, *topology (b)*, and *topology (c)*.

We compare the performance of the generated AAPC routine with *MPI_Alltoall* in LAM/MPI 6.5.9 and a recently improved MPICH 1.2.6 [19]. The generated routines use LAM/MPI point–to–point (send and recv) primitives. We will use the term GENERATED to denote the automatically generated routines. To make a fair comparison, we port the *MPI_Alltoall* routine in MPICH to LAM/MPI and use MPICH-LAM to denote the ported routine.

6.1. The generated routines and generation time

Table 1 shows the algorithms selected in the automatically generated routines for the topologies in Figure 2. For comparison purposes, the algorithms

Table 1. Algorithms in the generated routines

Topo.	Generated AAPC routine
(a)	Simple ($1 \leq n < 8718$) Pair with light barrier ($8718 \leq n < 31718$) Pair w. N=1 MPI barriers ($31718 \leq n < 65532$) Pair w. N=5 MPI barriers ($65532 \leq n < 72032$) Pair w. MPI barrier ($72032 \leq n$)
(b)	Bruck ($1 \leq n < 112$) Simple ($112 \leq n < 29594$) Ring w. light barrier ($29594 \leq n < 144094$) Ring w. N=1 MPI barriers ($144094 \leq n$)
(c)	Bruck ($1 \leq n < 112$) Simple ($112 \leq n < 581$) Ring ($581 \leq n < 8656$) Ring w. light barrier ($8656 \leq n < 65532$) Topology specific ($65532 \leq n$)

Table 2. Algorithms in LAM/MPI and MPICH

LAM/MPI	MPICH
Simple	Bruck ($1 \leq n \leq 256$) Spreading Simple ($257 < n \leq 32K$) Pair ($32K < n$ and p is a power of 2) Ring ($32K < n$ and p is not a power of 2)

in LAM/MPI and MPICH are depicted in Table 2. From Table 1, we can see that for different topologies, the optimal algorithms for AAPC are quite different, which indicates that the one-scheme-fits-all approach in MPICH and LAM cannot achieve good performance for different topologies. As will be shown in the performance study, while the MPICH AAPC algorithms are more sophisticated than the LAM/MPI AAPC algorithm, they perform worse for some topologies. This indicates that, to achieve good AAPC performance, the ability to adapt to the network topology is at least as important as finding good algorithms. The topology specific routine was selected for topology (c). For other topologies, using system wide synchronizations is effective in reducing network contention without knowing the network topology.

The time for generating the tuned routine is 1040 seconds for topology (a), 8913 seconds for topology (b), and 7294 seconds for topology (c). The tuning time depends on many factors such as the number of algorithms, the network topology, and how the performance is measured. The time is in par with that for other empirical approach based systems such as ATLAS [22]. Hence, like other empirical based systems, our approach can be used when the routine tuning time is relatively insignificant, e.g. when the application has a long execution time, or when the application needs to be executed repeatedly on the same system.

6.2. Performance

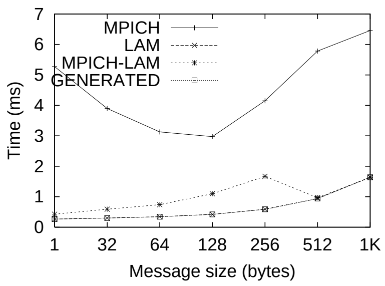
Figure 1 shows the code segment we use for measuring the performance. The number of iterations in each execution is varied according to the message size: more iterations are used for small message sizes to offset the clock inaccuracy. For the message ranges 1B-3KB, 4KB-12KB, 16KB-96KB, 128KB-384KB, and 512KB-, we use 100, 50, 20, 10, and 5 iterations, respectively. The results are the averages of three executions. We use the *average* time among all nodes as the performance metric. Before we present the results, we will point out two general observations in the experiments. For all example topologies:

1. Ignoring the minor inaccuracy in performance measurement, the generated AAPC routine never performs worse than the best of LAM, MPICH, and MPICH-LAM.
2. There exist some ranges of message sizes such that GENERATED out-performs each of LAM, MPICH-LAM, and MPICH by at least 40%.

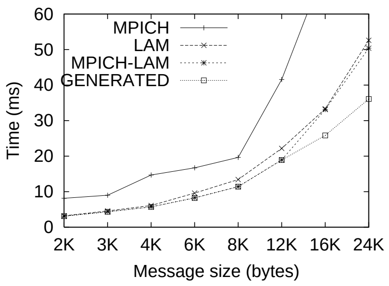
Figure 3 shows the performance on topology (a). For small messages ($1 \leq n \leq 256$), both LAM and GENERATED use the simple algorithm, which offers higher performance than the Bruck algorithm used in MPICH. When the message size is 512 bytes, MPICH changes to the spreading simple algorithm, which has similar performance to the simple algorithm. GENERATED, LAM, and MPICH-LAM have similar performance for message sizes in the range from 257 bytes to 9K bytes. As shown in Figure 3 (c), GENERATED significantly out-performs other schemes when the message size is larger than 16KB. For example, when the message size is 128KB, the time for GENERATED is 200.1ms and the time for MPICH (the best among LAM, MPICH, and MPICH-LAM) is 348.2ms, which constitutes a 74% speedup.

The results for topology (b) are shown in Figure 4. For this topology, the simple algorithm in LAM performs reasonably well for medium and large sized messages. In many cases, the simple algorithm performs better than the algorithms used in MPICH. GENERATED and LAM have similar performance for a very wide range of medium and large messages. For small messages, GENERATED offers higher performance than LAM in some cases and MPICH in other cases. For example, for the message size of 1 byte, the time for GENERATED and MPICH-LAM is 0.66ms while the time for LAM is 2.40ms. GENERATED improves over LAM by 264%. For the message size of 256 bytes, the time for GENERATED and LAM is 7.6ms while the time for MPICH-LAM is 11.2ms. GENERATED improves over MPICH-LAM by 47%.

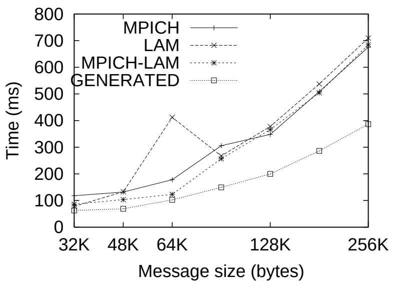
The results for topology (c) are shown in Figure 5. For small messages, the trend is similar to that for topology (b). For medium and large mes-



(a) Small message sizes

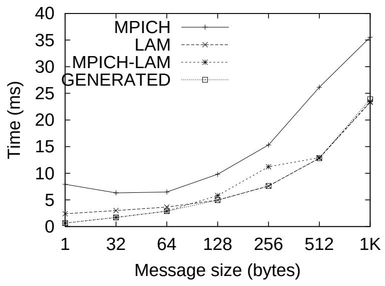


(b) Medium message sizes

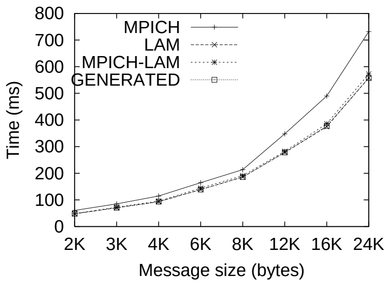


(c) Large message sizes

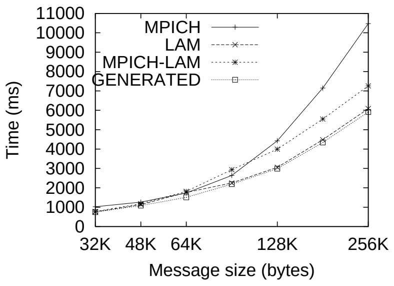
Figure 3. Performance for topology (a)



(a) Small message sizes



(b) Medium message sizes



(c) Large message sizes

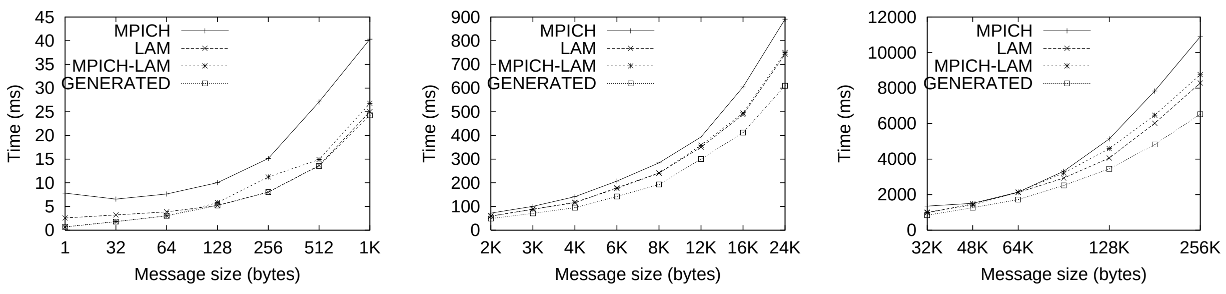
Figure 4. Performance for topology (b)

sages, GENERATED performs noticeably better than LAM, MPICH, and MPICH-LAM. For a very wide range of medium and large sized messages, GENERATED is about 10% to 35% faster than the best among LAM, MPICH, and MPICH-LAM. For example, for the message size of 32KB, the time for GENERATED is 842.7ms while the time for LAM (the best among LAM, MPICH, and MPICH-LAM) is 977.2ms: a 16% improvement. For the message size of 128KB, the time for GENERATED is 3451ms while the time for MPICH-LAM is 4590ms: a 33% improvement.

There are some interesting points to be noted in the performance results. First, while MPICH attempts to improve the AAPC performance by using more sophisticated AAPC algorithms, our experimental results do not give clear indications whether the MPICH AAPC routine is better than or worse than the simple AAPC implementation in LAM. This is because an AAPC algorithm achieves good performance only on particular platforms. Hence, to consistently achieve high performance on different platforms, it is essential that different AAPC algorithms are used. Such adaptability cannot be supported in conventional AAPC implementations, but can be provided with an empirical approach. Second, an AAPC implementation that takes into consideration some of the parameters may not be sufficient to yield good performance. MPICH attempts improve AAPC performance by considering some parameters. The AAPC routine in MPICH is able to adapt based on message sizes. However, as shown in the performance study, the performance of the MPICH AAPC routine is far from optimal. To consistently achieve high performance, the aggregated effects of all major system parameters must be considered. This problem is addressed in an empirical approach.

7. Conclusion

Traditional AAPC implementations, such as the ones in MPICH and LAM/MPI, suffer from an inherent limitation: the architecture specific information cannot be fully incorporated. This limitation is due to the fact that the routines are implemented before the program execution environment is decided. An empirical approach overcomes this limitation and can produce AAPC routines that are adaptable to different platforms. We report an automatic AAPC routine generator that is based on an empirical approach and demon-



(a) Small message sizes (b) Medium message sizes (c) Large message sizes

Figure 5. Performance for topology (c)

strate that an empirical approach is able to generate AAPC routines that offer higher performance than conventional AAPC implementations. It must be noted that different timing mechanisms can result in very different tuned routines. For the empirical approach to be effective, it is crucial to develop good timing mechanisms that accurately measure the performance.

References

[1] S. Bokhari. Multiphase Complete Exchange: a Theoretical Analysis. *IEEE Trans. on Computers*, 45(2), 1996.

[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 TPDS*, 8(11):1143-1156, Nov. 1997.

[3] H. G. Dietz, T. M. Chung, T. I. Mattox, and T. Muhammad. Purdue's Adapter for Parallel Execution and Rapid Synchronization: The TTL_PAPERS Design. *Technical Report*, Purdue University School of Electrical Engineering, January 1995.

[4] V. V. Dimakopoulos and N.J. Dimopoulos. Communications in Binary Fat Trees. In *IEEE ICDCS*, 1995.

[5] A. Faraj and X. Yuan. Message Scheduling for All–to–all Personalized Communication on Ethernet Switched Clusters. In *IEEE IPDPS*, April 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] W. Gropp and E. Lusk. Reproducible Measurements of MPI Performance Characteristics. *ANL/MCS-P755-0699*, Argonne National Labratory, June 1999.

[8] S.E. Hambrusch, F. Hameed, and A. A. Khokhar. Communication Operations on Coarse-Grained Mesh Architectures. *Parallel Computing*, Vol. 21, pp. 731-751, 1995.

[9] S. L. Johnsson and C. T. Ho. Optimum Broadcasting and Personalized Communication in Hypercubes. *IEEE Transactions on Computers*, 38(9):1249-1268, Sept. 1989.

[10] L. V. Kale, S. Kumar, K. Varadarajan. A Framework for Collective Personalized Communication. In *IEEE IPDPS*, April, 2003.

[11] LAM/MPI Parallel Computing. http://www.lam-mpi.org/.

[12] C. C. Lam, C. H. Huang, and P. Sadayappan. Optimal Algorithms for All–to–All Personalized Communication on Rings and two dimensional Tori. *JPDC*, 43(1):3-13, 1997.

[13] W. Liu, C. Wang, and K. Prasanna. Portable and Scalable Algorithms for Irregular All–to–all Communication. In *IEEE ICDCS*, 1996.

[14] The MPI Forum. *The MPI-2: Extensions to the Message Passing Interface*, July 1997.

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

[16] N.S. Sundar, D. N. Jayasimha, D. K. Panda, and P. Sadayappan. Hybrid Algorithms for Complete Exchange in 2d Meshes. *International Conference on Supercomputing*, pages 181–188, 1996.

[17] A. Tam and C. Wang. Efficient Scheduling of Complete Exchange on Clusters. In *ISCA PDCS*, Aug. 2000.

[18] R. Thakur and A. Choudhary. All-to-all Communication on Meshes with Wormhole Routing. *8th International Parallel Processing Symposium (IPPS)*, 1994.

[19] R. Thakur, R. Rabenseifner, and W. Gropp. Optimizing of Collective Communication Operations in MPICH. *ANL/MCS-P1140-0304*, Math. and Computer Science Division, Argonne National Laboratory, March 2004.

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

[21] E. A. Varvarigos and D. P. Bertsekas. Communication Algorithms for Isotropic Tasks in Hypercubes and Wraparound Meshes. *Parallel Computing*, Volumn 18, pages 1233-1257, 1992.

[22] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. In *SC'98: High Performance Networking and Computing*, 1998.