A TUNABLE COLLECTIVE COMMUNICATION FRAMEWORK ON A CLUSTER OF SMPS

Meng-Shiou Wu
Department of Electrical and Computer Engineering,
Iowa State University
SCL, Ames Laboratory, U.S. DOE
Ames, Iowa 50011, USA
mswu@iastate.edu

Ricky A. Kendall
Scalable Computing Laboratory
Ames Laboratory, U.S. DOE
Department of Computer Science,
Iowa State University
Ames, Iowa 50011, USA
rickyk@scl.ameslab.gov

Srinivas Aluru
Department of Electrical and Computer Engineering,
Department of Computer Science,
Iowa State University
Ames, Iowa 50011, USA
aluru@iastate.edu

ABSTRACT
In this paper we investigate a tunable MPI collective communications library on a cluster of SMPs. Most tunable collective communications libraries select optimal algorithms for inter-node communication on a given platform. We add another layer of intra-node communications composed by several tunable shared memory operations. We explore the advantages of our approach, and discuss when to use our approach, when to switch to another approach on the shared memory layer. Experimental results indicate that collective communications designed by such an approach with proper tuning can outperform vendor implementations.

KEY WORDS
Cluster Computing, Collective Communications, MPI, Shared Memory Operations, Tunable Libraries

1 Introduction

A cluster that consists of SMP nodes provides an attractive architecture for improving MPI communications. Within an SMP node, communications between processes can directly use shared memory without going through the communication network. Since collective communications are usually designed using send and receive operations, a traditional method for improving collective communications on SMP nodes is by optimizing the particular send and receive operations. However, these approaches do not take advantage of the full potential of SMP architectures. Several attempts have been made during the last few years to use the SMP architecture for improving some collective operations [14, 16]. Most approaches are programming oriented or specific to a particular platform. It is not clear whether the proposed approaches can be extended to all MPI collective operations or implemented on a generic platform.

The question is complicated further when considering an automatically tuned collective communication library on a cluster of SMPs. Due to the diversity of parallel computing architectures, optimization of collective communications on one platform does not guarantee optimal performance on other platforms. There are at least two collective communications libraries with automatically tuned functionality [10, 17]. These libraries consider only clusters composed of single processor nodes. While many modern clusters are made up of SMP nodes, we are not aware of an automatically tuned MPI library that takes the SMP node architecture into account.

Several questions arise when designing an automatically tuned collective communications library on a cluster of SMPs: What is the model that provides tunable parameters for SMP collective communications? When are optimizations of send and receive enough to provide good performance on SMP clusters? Does a good algorithm on a single processor node based cluster also perform well on SMP based clusters? There are several approaches available to take advantage of the SMP architectures, when should we switch from one approach to another?

As the first step for the design of such a library on SMP clusters, we use a generic model of basic operations. Based on this model and the characteristics of SMP architectures, we decompose collective communications into several tunable shared-memory operations. Several major collective communications are constructed from these basic operations. We explore the advantages of our approach, and discuss the analytical switching criteria to other approaches. We then show how we implement our collective communications, and compare our experimental results with vendor MPI implementations.

2 Related Work

Several previous attempts to use shared memory for collective communications are described as follows: IBM MPI optimizes send and receive for shared memory, as well as collective operations which are built on top of these two operations. Sistare *et al.* [14] use MPI operations for inter-node communications and shared memory for intra-node communication to design collective communications. Tipparaju *et al.* [16] use LAPI for inter-node communication instead, which make it possible to overlap shared memory operations with inter-node communication. These two efforts present algorithms for broadcast, reduce, and barrier. More complex operations such as scatter, gather, all-to-all, etc., are not considered. Golebiewski and co-workers [7] design these more complex operations for SMP nodes with only two processors per node. Gropp *et al.* [8] modified MPICH to work on shared-memory vector supercomputers. Lumetta *et al.* [11] designed multi-protocol Active Messages for communication on SMP clusters. Bader and JáJá designed SIMPLE [1], a methodology for programming on SMP clusters, with library support for broadcast, reduce and barrier on SMP nodes.

Automatically tuned collective communications library include the work of Fagg and co-workers who designed ACCT [6, 17] that automatically tune collective communications on a cluster, and Huse [10] designed mechanisms to automatically tune MPI collective communications on clusters connected by a wide area network. Both were based on the LogP [5] model to construct optimal communication trees for inter-node communications.

3 Basic Model

Our tunable collective communication model consists of two levels: the inter-node network communication and the intra-node shared-memory operations. For communication between nodes we use standard MPI send and receive operations as a generic base. For intra-node shared-memory operations, we use the following model that can be implemented on any system with System V shared-memory functionality: a shared-memory segment of limited size is allocated for communications between any two or more processors on the same node. Any communications within an SMP node are begun by the source process copying data from its local memory into the shared-memory segment. The receiving processes then copy data from the shared-memory segment to their local memories. Within each SMP node, one processor (group_comm) is in charge of scheduling communications with other nodes. Since we are targeting a generic model we assume there is no overlap between the two levels of communications.

3.1 Test Platform

Our test platform is the IBM SP system at the National Energy Research Scientific Computing Center. It is a distributed-memory parallel supercomputer with 380 compute nodes. Each node has 16 POWER3+ processors and at least 16 GBytes of memory, thus at least 1 GByte of memory per processor. Each node has 64 KBytes of L1 data cache and 8192 KBytes of L2 cache. The nodes are connected to each other with an IBM proprietary switching network. The IBM MPI implementation has optimizations for send and receive operations on shared memory which can be turned on or off. There are no optimizations for collective operations.

All the figures in this paper use the notation *A*x*B* to denote that the experiment uses *A* nodes, each with *B* MPI tasks. For example, 4x8 means we are using 4 nodes, each with 8 MPI tasks for the specific experiment. The curves labeled with SHM mean we used our shared-memory implementation. Those labeled with XCC mean we utilized a combination of MPI for inter-node communications and shared-memory operations for intra-node communications.

4 Collective Communications on the Distributed Memory Layer

Many collective communication algorithms can be found in the literature such as CCL by Bala and co-workers [2], InterCom by Barnett and co-workers [3, 4], and the work of Mitra *et al.* on a fast collective communication library [12]. On distributed memory layer, we implemented the binomial tree algorithm and flat trees for all four implemented collective communications. We also use scatter-allgather broadcast algorithm as an example of a two stage algorithm. We import it from MPICH 1.2.5 [15] with a small modification to make it work on the IBM SP but keeping the basic algorithm intact. The tuning criteria on this layer is straightforward: find the algorithm with the best performance for a particular operation. Theoretical analysis does not always give the answer; for example, the sequential flat tree algorithm can perform better than binomial tree algorithm for some operations. The best algorithm for an operation on a certain platform usually has to be found through experimentation.

5 Collective Communications on Shared Memory Layer

Based on the shared-memory communication model described above, we define the following basic shared-memory operations on a given SMP node:

(1) ***Sender_put()*** *: Sender puts message into a shared buffer.*
(2) ***Receiver_get()*** *: Receiver gets message from a shared buffer.*
(3) ***Pair_sync()*** *: Synchronization between two processes.*
(4) ***Group_get(p,m)*** *: A group of p processes gets a message of size m from a shared buffer.*
(5) ***Group_seg_get(p,m)*** *: A group of p processes gets a message from a shared buffer, each process gets its part of the data of size m.*
(6) ***Group_seg_put(p,m)*** *: A group of p processes puts a message into a shared buffer, each process puts its part of the data of size m.*
(7) ***Group_sync(p)*** *: Synchronization among a group of p processes.*

(*Sender_put(), Pair_sync(), Receiver_get()*) is the minimum set of operations required for implementing collective communications on a shared-memory layer within the SMP node. A send and receive operation between processes in message passing can be replaced by (*Sender_put(), Pair_sync(), Receiver_get()*) within an SMP node. Some MPI implementations on the shared-memory layer also focus on optimized shared-memory send and receive operations since most MPI implementations implement collective communications based on the send and receive functionality. However, this minimum set of operations does not take advantage of concurrent memory access on the shared-memory layer, and in many scenarios does not give optimal performance. For this reason, we have added several operations that are specific to shared memory and decompose collective communications into these basic shared-memory operations.

Our decomposition is based on the following observations: Figure 1 shows the performance of two broadcast approaches to access a data block of 8K. One approach uses the generic shared-memory operations delineated in this work. This is one stage of (*Sender_put(), Group_sync(), Group_get()*). The second approach uses the vendor-based implementation and it is *logp* stages of shared-memory send and receive, *p* is the number of processors used within an SMP node. Shared memory operations performed better than *logp* stages of send and receive. When the number of processes increase, the rate of run time increase is also much less than the second approach. However, this advantage can be used only up to a certain data size. When the data size increases, the hidden cost of (*Sender_put(), Group_sync()* , *Group_get()*) such as page faults, TLB misses, and cache coherence maintenance also increases and this approach loses its advantage. Our decomposition is used to find the best buffer size for shared-memory operations and use them whenever optimal.

In Table 1, we outline four major collective communication operations we implemented using the above basic shared-memory operations.

This is certainly not the only way to decompose collective communications. If we assume there are "hot spots" as mentioned in [14], we may want to develop different algorithms to avoid concurrent access of certain portions of memory. For example, this may be important in the hierarchical NUMA memory on an SGI origin system. Our current approach is to find the maximum size that can take advantage of concurrent memory access, so we do not take "hot spots" into consideration.

5.1 Analytical Tuning Criteria

There are three mechanisms for communication within an SMP node: our shared memory approach, using the vendor send/receive shared-memory operations, and using communication network. We give our theoretical analysis here as the guideline to indicate when to use one approach and when to switch to another one.

Assume α is the startup latency, β is the inverse transmission rate, *m* is the message size, *A* is the number of SMP nodes in a cluster, *B* is the number of processors per SMP node, and *p* is the total number of processors.

The inter-node communication cost between two nodes is $(\alpha + \beta * m)$. If an inter-node collective communication is done by binomial tree algorithm, then its cost is $logA * (\alpha + \beta * m)$; if it is done using the flat tree algorithm, the inter-node communication cost is $A * (\alpha + \beta * m)$. Within an SMP node, the communication cost through the network is $logB * (\alpha + \beta * m)$ using binomial tree algorithm, and $B * (\alpha + \beta * m)$ using flat tree algorithm. If intra-node communication is done by using the shared-memory send and receive, then the communication cost is *logB*, or *B* stages of shared-memory send and receive. If concurrent memory access to shared memory is possible, then the intra-node communication latency is *k* stages of (*group_op(single_op)* + *sync_op* + *group_op(single_op)*). The value of *k* depends on *m* and how a collective communication is implemented on the shared-memory layer. Since we assume there is no overlapping between inter-node and intra-node communication, the total cost of a collective communication is just the sum of communication costs on the two layers.

Assuming binomial algorithm is used, then the choice of a particular approach basically depends on the relative cost of (1) $logB * (\alpha + \beta * m)$, (2) *logB* stages of shared-memory send and receive, or (3) *k* stages of *group_op(single_op)* + *sync_op* + *group_op(single_op)*, Clearly, when the data size is small enough that *k* = 1, or when we can use group operations to access the data such as in the broadcast algorithm, (3) is the best choice. If *B* = 2, or we can only use send and receive on the shared-memory layer such as in the scatter or gather algorithm with the large data size, then optimized shared-memory send/receive (2) will certainly help. When communication through the network is faster than through shared memory, (2) can be replaced by (1). If network communication is so fast that when the data size is small, even the cost of $logB * (\alpha + \beta * m)$ is smaller than one stage of *group_op(single_op)* + *sync_op* + *group_op(single_op)*, all that is needed is (1).

It is clear from the above analysis that taking advantages of SMP architecture can have performance gains when the data size is small to medium (when *k* = 1). Vetter conducted experiments on several large-scale scientific applications [18] and observed, "the payload size of these collective operations is very small and this size remains practically invariant with respect to the problem size or the number of tasks." Based on the Vetter's observation, we believe that this approach should be considered as a way to improve the performance of MPI collective communications on SMP based clusters.

It is difficult to come up with a theoretical formulation to predict the relative performance of (1), (2) and (3). At this point we have chosen to compare them by experimentation so that we can shed more light on an analytical functional form to determine when to switch from one algorithm to another.

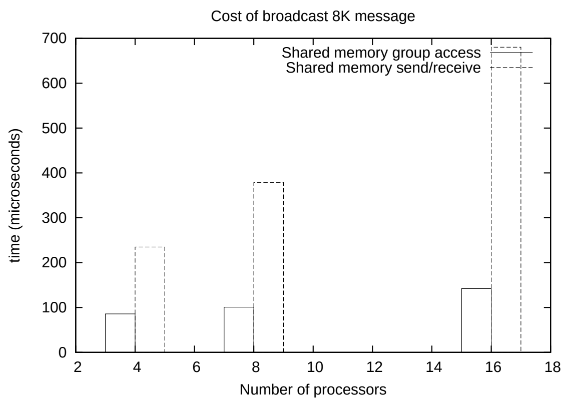


Figure 1. Cost of accessing data within an SMP node

C.C.	***(a) m < B***	***(b) m > B***
1 Broadcast	*Sender_put(), Group_sync(), Group_get()*	*Sender_put(), Group_sync(), Group_get() with pipeline*
2 Scatter	*Sender_put(), Group_sync(), Group_seg_get()*	*Sender_put(), Pair_sync(), Receiver_get() with pipeline*
3 Gather	*Group_seg_put(), Group_sync(), Receiver_get()*	*Sender_put(), Pair_sync(), Receiver_get() with pipeline*
4 All-to-all	*Group_seg_put(), Group_sync(), Group_seg_get()*	*pairwise-[Sender_put(), Pair_sync(), Receiver_get()] with pipeline*

Table 1. Four collective communications decomposed into basic shared-memory operations.*m* is the total communication message size, and *B* is shared buffer size.

6 Tuning Parameters on Shared Memory Layer

Based on the analysis, our turning strategy is to find the best buffer size such that a collective communication can be completed in one stage. When message size is larger than this buffer size, find the proper number of pipeline buffer to obtain better performance. When pair wise communications is the best choice for a collective communication, we switch to vendor's send/receive implementations.

6.1 Synchronization Scheme

We have tested different synchronization methods and found out that polling with no-op generally gives a good performance. The experimental results in this paper all use polling with no-op.

6.2 Shared Buffer Size

With a huge buffer size the collective communication may be able to be completed in one stage, but hidden costs such as TLB misses may degrade performance. If we use a small buffer size, the sender and receiver may spend too much time in synchronization. To find a proper buffer size, we measured the cost of Sender_put(). We tested data sizes from 1K, 2K, 4K up to 8MB, and for each data size we repeatedly copied data segments to buffer sizes of 16, 32, 64 bytes up to the test data size. Figure 2 shows the cost of copying data sizes of 512K to 8MB to shared memory through different shared-buffer sizes using log-log scale. The curve indicates a buffer from 4K to 16K is best for copying 8MB on our test system. We chose 8K as the shared-buffer size which generally gives a good performance.

6.3 Pipeline buffers

When multiple buffers are used in a pipelined fashion, the number of buffers used affects performance. While two buffers are frequently suggested in the literature, it does not guarantee optimal performance. The left panel of figure 3 shows how we tuned buffer numbers for broadcast operations. Two buffers of size 8K do not perform as well as 2 buffers of size 16K. When we increase to 16 buffers, an 8K buffer size outperforms the other combinations for broadcast. On the right panel of figure 3 is the performance of the same implementation, with and without tuning, against the vendor supplied MPI_Bcast() function. Without tuning (2 buffers of size 4K), the run time is 30% slower than the vendor implementation. After tuning, the run time is 50%

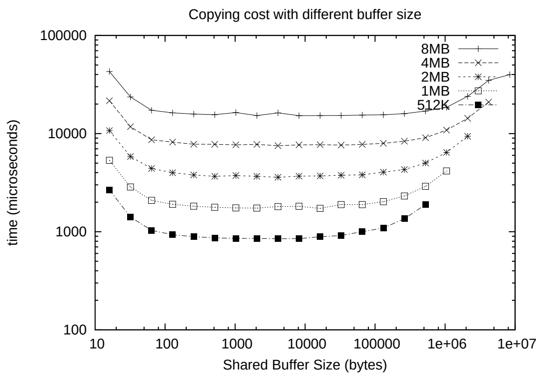
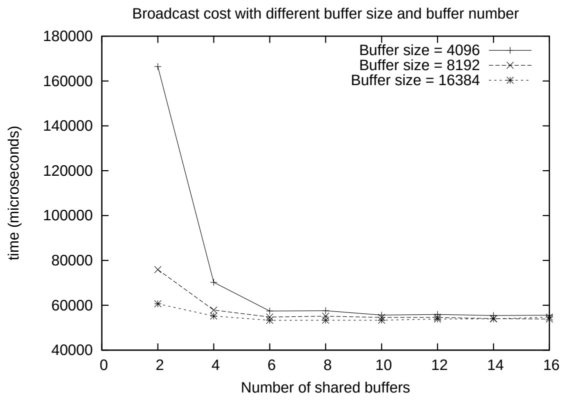


Figure 2. Shared buffer size and copying performance



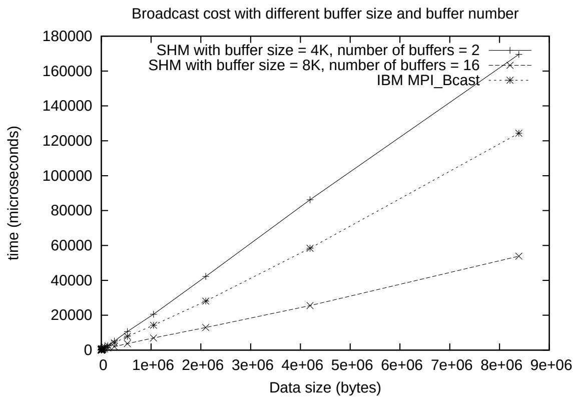


Figure 3. Performance of broadcast as a function of buffers

faster than the vendor MPI_Bcast.

7 Performance Measurement

We use the following steps to measure performance of a single operation:
(1) Purge cache
(2) Barrier call
(3) Start operation timing
(4) Execute collective operation
(5) End operation timing
(6) Allreduce operation to extract the node with the longest run time

The experiments on each operation were performed many times the average is computed. Purging cache is necessary to ensure that data comes from main memory, not from cache. To purge cache, a memory block of L2 cache size is allocated; each byte in the block is then set to 0. This makes sure that no data used for communication is present in cache.

Each node starts timing after barrier synchronization. When the collective operation is finished, the node with the longest run time is identified. By doing this we can meet the condition "between the first process starting and the last process finishing the collective operation" as described by Worsch and co-workers [19]. Several issues relating to benchmarking MPI collective operations can be found in Natawut and Lionel [13], Worsch and co-workers SKaMPI [19], and the perftest of Gropp and Lusk for MPICH [9].

8 Experimental Results

8.1 Broadcast

Figure 4 shows the performance of our generic hierarchical implementation of broadcast against vendor's MPI_Bcast. Our approach is to select binomial tree algorithm for inter-node communication, and use shared memory operations for intra-node communications. The result is usually at least 30% faster than MPI_Bcast, and sometimes more than 50% faster than the MPI_Bcast time, especially when the data size is small.

8.2 Scatter and Gather

Our selection for scatter is as follow: when message size is less than or equal to 128K, use shared memory opera-

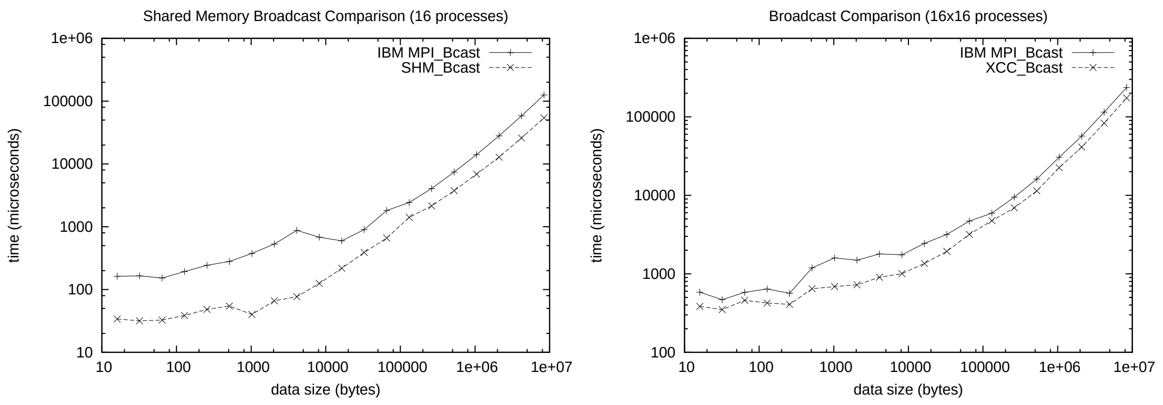


Figure 4. Performance of our broadcast implementation versus the vendor supplied broadcast on one SMP node and 16 SMP nodes

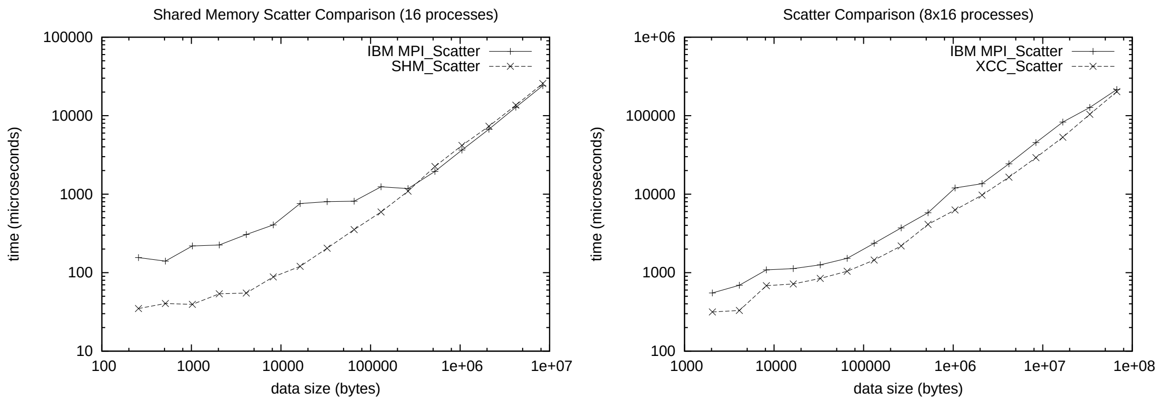


Figure 5. Performance of our scatter implementation versus the vendor supplied broadcast on one SMP node and 8 SMP nodes

tions; if message is larger than 128K, switch to flat tree algorithm that implemented using vendor's send/receive implementations. The results (c.f. Figure 5) demonstrate that when the total data size is less than 128K, the run time of the concurrent group access is just about 40% faster than MPI_Scatter on a single node with 16 MPI tasks. When using several nodes with 16 MPI tasks per node, our implementation can be as much as 30% faster than MPI_Scatter. Our shared-memory implementation performs better than the vendor implementation from 8K to 128K message size due to less synchronizations are required for shared memory operations at this range. Gather is just the inverse of scatter and the performance with the appropriate operations is similar.

8.3 All-to-all

Figure 6 shows the performance comparison of two shared-memory all-to-all approaches. Within an SMP node, the shared-memory all-to-all algorithm performs better than the other approaches when the data size is small to medium. However, our approach cannot perform better than the pair-wise exchange on the testing platform even when data size is small. When we are not using the pair-wise exchange algorithm and assuming only one process in each node is in charge of communication with other nodes, we have to do a shared-memory gather, inter-node all-to-all, and a shared-memory scatter set of steps. Even when the data size is small, group_comm has to re-arrange data so that shared-memory gather and scatter can be done in one stage; otherwise it is *B* stages of the shared-memory scatter or gather. This extra cost of memory operations are larger than the performance gain of the shared-memory all-to-all algorithm. Our choice here is to use pair-wise exchange when all-to-all operation involves inter-node communications, and use shared memory operations when only one SMP node is used.

8.4 Two stage algorithms

Barnett and co-workers' broadcast algorithm [3] uses two stages, first message segments are scattered to all processes, then each process collects data by Allgather. The algorithm works well with large data sizes and is im-

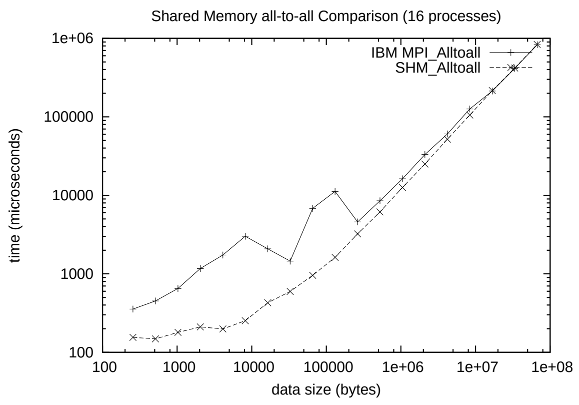


Figure 6. Performance of our all-to-all implementation versus the vendor supplied all-to-all on one SMP node

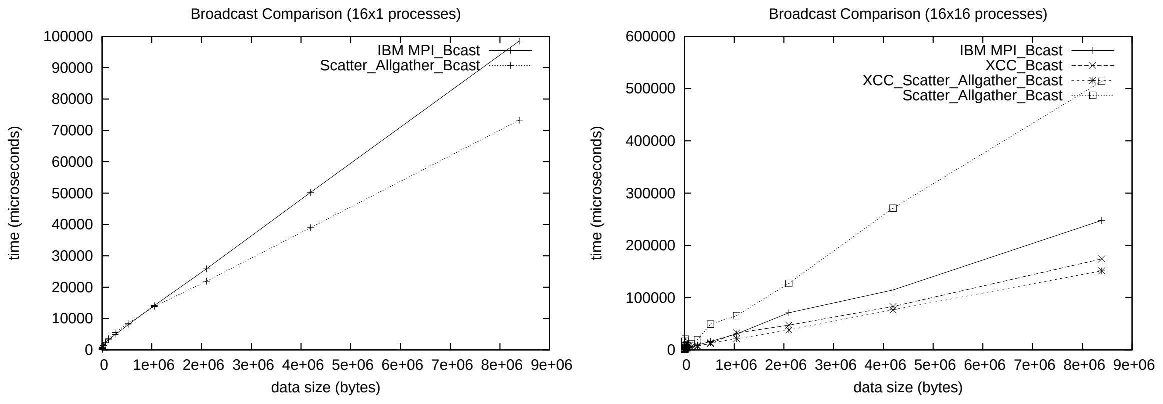


Figure 7. Performance of two stage broadcast of MPICH implementation versus the vendor supplied broadcast and our modified two stage broadcast versus the vendor supplied broadcast on 16 SMP nodes

plemented in MPICH 1.2.5. The left panel of Figure 7 shows the performance of the MPICH implementations of this algorithm compared to the IBM MPI_Bcast on large data sizes when communication involves only inter-node messages. However, if intra-node communication is optimized only with send/receive, the algorithm performance degrades due to extra memory copies. The right panel of Figure 7 shows the same algorithm run on 16 nodes with 16 MPI tasks; the performance of this algorithm is much worse than IBMs MPI_Bcast. We modified the algorithm so that it works only on the inter-node level. On the intra-node level we use shared-memory operations which incur no extra memory copy, and this implementation even outperforms our broadcast implementation using a binomial tree for inter-node communication.

9 Conclusion and Future Work

To achieve optimal communication performance on a cluster of SMP nodes, not only do we need optimal algorithm for inter node communication, but also good shared memory communication schemes that can be adjusted according to different collective communication characteristics. By decomposing collective communications into shared-memory operations, we provide another approach to improve collective communications within an SMP node. Our experimental results show that our approach can utilize SMP clusters better than vendor's implementations. We are currently building an automatically tuned collective communications library based on the results.

Future work besides an automatically tuned library includes: (a) mechanisms for automatic tuning and reduction of experiment times, (b) investigation of generic methods to overlap inter- and intra-node communications, and (c) incorporation of "hot spots" or other shared-memory characteristics to improve shared-memory communication performance.

10 Acknowledgements

This work was performed under auspices of the U. S. Department of Energy under contract W-7405-Eng-82 at Ames Laboratory operated by the Iowa State University of Science and Technology. Funding was provided by the

Mathematical, Information and Computational Science division of the Office of Advanced Scientific Computing Research. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

References

[1] David A. Bader and Joseph JáJá. SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors. *Journal of Parallel and Distributed Computing*, pages 92–108, 1999.

[2] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.-T. Ho, S. Kipnis, and M. Snir. A Portable and Tunable Collective Communication Library for Scalable Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, pages 154–164, 1995.

[3] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor Collective Communication Library (InterCom). In *Scalable High Performance Computing Conference*, pages 357–364, 1994.

[4] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts. Building a HighPerformance Collective Communication Library. In *Proceedings of Supercomputing*, pages 107–116, 1994.

[5] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[6] Graham E. Fagg, Sathish S. Vadhiyar, and Jack J. Dongarra. ACCT: Automatic Collective Communications Tuning. In *EuroPVM/MPI*, 2000.

[7] Maciej Golebiewski, Rolf Hempel, and Jesper Larsson Träff. Algorithms for Collective Communication Operations on SMP Clusters. In *The 1999 Workshop on Cluster-Based Computing held in conjunction with 13th ACM-SIGARCH International Conference on Supercomputing*, 1999.

[8] William Gropp and Ewing Lusk. A High-Performance MPI Implementation on A Shared-Memory Vector Supercomputer. *Parallel Computing*, pages 1513–1526, 1997.

[9] William Gropp and Ewing Lusk. Reproducible Measurements of MPI Performance Characteristics. In *Recent advances in parallel virtual machine and message passing interface: 6th European PVM/MPI Users' Group Meeting*, pages 26–29, 1999.

[10] Lars Paul Huse. Collective Communication on Dedicated Clusters of Workstations. In *European PVM/MPI*, 1999.

[11] Steve Lumetta, Alan Mainwaring, and David Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of Supercomputing*, San Joe, USA, 1997.

[12] P. Mitra, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Fast Collective Communication Libraries, Please. In *Proceedings of the Intel Supercomputing User's Group Meeting*, 1995.

[13] Natawut Nupairoj and Lionel M. Ni. Performance Metrics and Measurement Techniques of Collective Communication Services. In *First International Workshop on Communication and Architectural Support for Network-Based Parallel Computing*, pages 212–226, 1997.

[14] Steven Sistare, Rolf van de Vaart, and Eugene Loh. Optimization of MPI Collectives Clusters of Large-Scale SMP's. In *Proceedings of Supercomputing*, 1999.

[15] Rajeev Thakur and William Gropp. Improving the Performance of Collective Operations in MPICH. In *European PVM/MPI*, 2003.

[16] Vinod Tipparaju, Jarek Nieplocha, and Dhabaleswar K. Panda. Fast Collective Operations Using Shared and Remote Memory Access Protocols on Clusters. In *International Parallel and Distributed Processing Symposium*, 2003.

[17] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically Tuned Collective Communications. In *Proceedings of Supercomputing*, 2000.

[18] Jeffrey S. Vetter and Frank Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *International Parallel and Distributed Processing Symposium*, 2002.

[19] Thomas Worsch, Ralf H. Reussner, and Werner Augustin. Benchmarking Collective Operations. In E. Krause and W. Jäger, editors, *High Performance Computing in Science and Engineering 2002*, Transactions of the High Performance Computing Center Stuttgart (HLRS). 2002.