Improving the Performance of Collective Operations in MPICH

Rajeev Thakur and William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439, USA
{thakur, gropp}@mcs.anl.gov

Abstract. We report on our work on improving the performance of collective operations in MPICH on clusters connected by switched networks. For each collective operation, we use multiple algorithms depending on the message size, with the goal of minimizing latency for short messages and minimizing bandwidth usage for long messages. Although we have implemented new algorithms for all MPI collective operations, because of limited space we describe only the algorithms for allgather, broadcast, reduce-scatter, and reduce. We present performance results using the SKaMPI benchmark on a Myrinet-connected Linux cluster and an IBM SP. In all cases, the new algorithms significantly outperform the old algorithms used in MPICH on the Myrinet cluster, and, in many cases, they outperform the algorithms used in IBM's MPI on the SP.

1 Introduction

Collective communication is an important and frequently used component of MPI and offers implementations considerable room for optimization. MPICH, although widely used as an MPI implementation, has until now had fairly rudimentary implementations of the collective operations. We have recently focused on improving the performance of all the collective operations in MPICH. Our initial target architecture is the one that is the most popular among our users, namely, clusters of machines connected by a switch, such as Myrinet or the IBM SP switch. For each collective operation, we use multiple algorithms based on message size: The short-message algorithms aim to minimize latency, and the long-message algorithms aim to minimize bandwidth usage. Our approach has been to identify the best algorithms known in the literature, improve on them where possible, and implement them efficiently. Our implementation of the algorithms handles derived datatypes as well as non-power-of-two number of processes.

We have implemented new algorithms in MPICH for all the collective operations, namely, scatter, gather, allgather, broadcast, reduce, allreduce, reduce-scatter, scan, and barrier. Because of limited space, however, we describe only the algorithms for allgather, broadcast, reduce-scatter, and reduce. We use the

SKaMPI benchmark [19] to measure the performance of the algorithms on two platforms: a Linux cluster at Argonne connected with Myrinet 2000 and the IBM SP at the San Diego Supercomputer Center. On the Myrinet cluster we use MPICH-GM and compare the performance of the new algorithms with the old algorithms in MPICH-GM. On the IBM SP, we use IBM's MPI and compare the performance of the new algorithms with the algorithms used in IBM's MPI. On both systems, we ran one MPI process per node. We implemented the new algorithms as functions on top of MPI point-to-point operations, so that we can compare performance simply by linking or not linking the new functions.

The rest of this paper is organized as follows. Section 2 describes related work in this area. Section 3 describes the cost model we use to guide the selection of the algorithms. The algorithms and their performance are described in Section 4. We conclude in Section 5 with a brief discussion of future work.

2 Related Work

Early work on collective communication focused on developing optimized algorithms for particular architectures, such as hypercube, mesh, or fat tree, with an emphasis on minimizing link contention, node contention, or the distance between communicating nodes [2–4, 13]. More recently, Dongarra et al. have developed automatically tuned collective communication algorithms [18]. Their approach consists of running tests to measure system parameters and then tuning their algorithms for those parameters. Researchers in Holland and at Argonne have optimized MPI collective communication for wide-area distributed environments [7, 8]. In such environments, the goal is to minimize communication over slow wide-area links at the expense of more communication over faster local-area connections. Research has also been done on developing collective communication algorithms for clusters of SMPs [12, 15–17], where communication within an SMP is done differently from communication across a cluster. Some efforts have focused on using different algorithms for different message sizes, such as the work by Van de Geijn et al. for the Intel Paragon [1, 9, 14], by Rabenseifner on reduce and allreduce [11], and by Kale et al. on all-to-all communication [6].

3 Cost Model

We use a simple model to estimate the cost of the collective communication algorithms in terms of latency and bandwidth usage and to guide the selection of algorithms for a particular collective communication operation. We assume that the time taken to send a message between any two nodes can be modeled as $\alpha + n\beta$, where α is the latency (or startup time) per message, independent of message size, β is the transfer time per byte, and n is the number of bytes transferred. We assume further that the time taken is independent of how many pairs of processes are communicating with each other, independent of the distance between the communicating nodes, and that the communication links are bidirectional (that is, a message can be transferred in both directions on the link in the same time as in one direction). The node's network interface is assumed to be single ported; that is, at most one message can be sent and one message can be received simultaneously. In the case of reduction operations, we assume that γ is the computation cost per byte for performing the reduction operation locally on any process.

4 Algorithms

In this section we describe the new algorithms and their performance.

4.1 Allgather

`MPI_Allgather` is a gather operation in which the data contributed by each process is gathered on all processes, instead of just the root process as in `MPI_Gather`.

The old algorithm for allgather in MPICH uses a ring method in which the data from each process is sent around a virtual ring of processes. In the first step, each process i sends its contribution to process $i+1$ and receives the contribution from process $i-1$ (with wrap-around). From the second step onwards each process i forwards to process $i+1$ the data it received from process $i-1$ in the previous step. If p is the number of processes, the entire algorithm takes $p-1$ steps. If n is the total amount of data to be gathered on each process, then at every step each process sends and receives $\frac{n}{p}$ amount of data. Therefore, the time taken by this algorithm is given by $T_{ring} = (p-1)\alpha + \frac{p-1}{p}n\beta$. Note that the bandwidth term cannot be reduced further because each process must receive $\frac{n}{p}$ data from $p-1$ other processes. The latency term, however, can be reduced if we use an algorithm that takes $\lg p$ steps. We use such an algorithm, called recursive doubling, for the new allgather in MPICH. Recursive doubling has been used in the past, particularly on hypercube systems; it is also used in the implementation of allreduce in [11].

Figure 1 illustrates how the recursive doubling algorithm works. In the first step, processes that are a distance 1 apart exchange their data. In the second step, processes that are a distance 2 apart exchange their own data as well as the data they received in the previous step. In the third step, processes that are a distance 4 apart exchange their own data as well the data they received in the previous two steps. In this way, for a power-of-two number of processes, all processes get all the data in $\lg p$ steps. The amount of data exchanged by each process is $\frac{n}{p}$ in the first step, $\frac{2n}{p}$ in the second step, and so forth, up to $\frac{2^{\lg p-1}n}{p}$ in the last step. Therefore, the total time taken by this algorithm is $T_{rec\_dbl} = \lg p\,\alpha + \frac{p-1}{p}n\beta$.

The recursive-doubling algorithm is straightforward for a power-of-two number of processes but is a little tricky to get right for a non-power-of-two number of processes. We have implemented the non-power-of-two case as follows. At each step of recursive doubling, if the current subtree is not a power of two, we do additional communication to ensure that all processes get the data they would

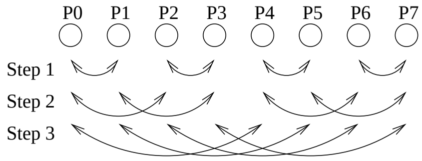

Fig. 1. Recursive doubling for allgather

have gotten had the subtree been a power of two. This communication is done in a logarithmic fashion to minimize the additional latency. This approach is necessary for the subsequent steps of recursive doubling to work correctly. The total number of steps for the non-power-of-two case is bounded by $2\lfloor \lg p \rfloor$.

Note that the bandwidth term for recursive doubling is the same as for the ring algorithm, whereas the latency term is much better. Therefore, one would expect recursive doubling to perform better than the ring algorithm for short messages and the two algorithms to perform about the same for long messages. We find that this situation is true for short messages (see Figure 2). For long messages (> 512 KB), however, we find that recursive doubling runs much slower than the ring algorithm, as shown in Figure 3. We believe this difference is because of the difference in the communication pattern of the two algorithms: The ring algorithm has a nearest-neighbor communication pattern, whereas in recursive doubling, processes that are much farther apart communicate. To confirm this hypothesis, we used the *b_eff* MPI benchmark [10], which measures the performance of about 48 different communication patterns, and found that, for long messages on both the Myrinet cluster and the IBM SP, some communication patterns (particularly nearest neighbor) achieve more than twice the bandwidth of other communication patterns. In MPICH, therefore, we use the recursive-doubling algorithm for short- and medium-size messages (< 512 KB) and the ring algorithm for long messages (≥ 512 KB).

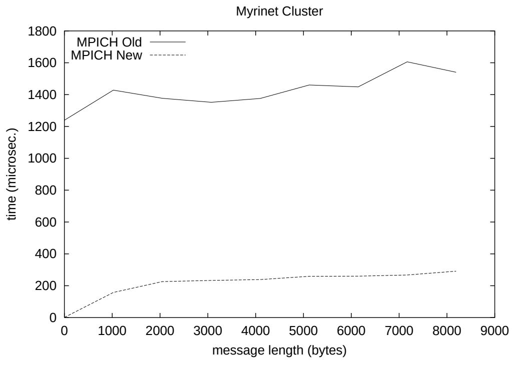

Fig. 2. Performance of allgather for short messages on the Myrinet cluster (64 nodes). The size on the x-axis is the total amount of data gathered on each process.

We note that the dissemination algorithm [5] used to implement barrier operations is a better algorithm for non-power-of-two number of processes as it takes $\lceil \lg p \rceil$ steps. In each step k of this algorithm, process i sends a message to process $(i + 2^k)$ and receives a message from process $(i - 2^k)$ (with wrap-around). This algorithm works well for barrier operations, where no data needs to be

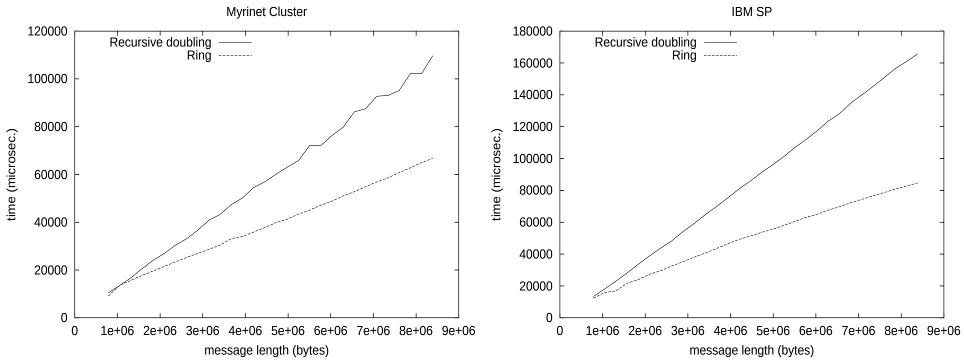

Fig. 3. Ring algorithm versus recursive doubling for long-message allgather (64 nodes). The size on the x-axis is the total amount of data gathered on each process.

communicated: Processes simply send 0-byte messages. We, however, find it not easy to use for collective operations that have to communicate data, because keeping track of which data must be routed to which process is nontrivial in this algorithm and requires extra bookkeeping and memory copies. With recursive doubling, on the other hand, keeping track of the data is trivial.

4.2 Broadcast

The old algorithm for broadcast in MPICH is the commonly used binary tree algorithm. In the first step, the root sends data to process (root + $\frac{p}{2}$). This process and the root then act as new roots within their own subtrees and recursively continue this algorithm. This communication takes a total of $\lceil \lg p \rceil$ steps. The amount of data communicated by a process at any step is n. Therefore, the time taken by this algorithm is $T_{tree} = \lceil \lg p \rceil (\alpha + n\beta)$.

This algorithm is good for short messages because it has a logarithmic latency term. For long messages, however, a better algorithm has been proposed by Van de Geijn et al. that has a lower bandwidth term [1, 14]. In this algorithm, the message to be broadcast is first divided up and scattered among the processes, similar to an `MPI_Scatter`; the scattered data is then collected back to all processes, similar to an `MPI_Allgather`. The time taken by this algorithm is the sum of the times taken by the scatter, which is $(\lg p\ \alpha + \frac{p-1}{p} n\beta)$ for a binary tree algorithm, and the allgather for which we use either recursive doubling or the ring algorithm depending on the message size. Therefore, for very long messages where we use the ring allgather, the time taken by the broadcast is $T_{vandegeijn} = (\lg p + p - 1)\alpha + 2\frac{p-1}{p} n\beta$.

Comparing this time with that for the binary tree algorithm, we see that for long messages (where the latency term can be ignored) and when $\lg p > 2$ (or $p > 4$), the Van de Geijn algorithm is better than binary tree. The maximum improvement in performance that can be expected is $(\lg p)/2$. In other words, the larger the number of processes, the greater the expected improvement in performance. Figure 4 shows the performance for long messages of the new algorithm versus the old binary tree algorithm in MPICH as well as the algorithm used by

IBM's MPI on the SP. In both cases, the new algorithm performs significantly better. Therefore, we use the binary tree algorithm for short messages (< 12 KB) and the Van de Geijn algorithm for long messages (≥ 12 KB).

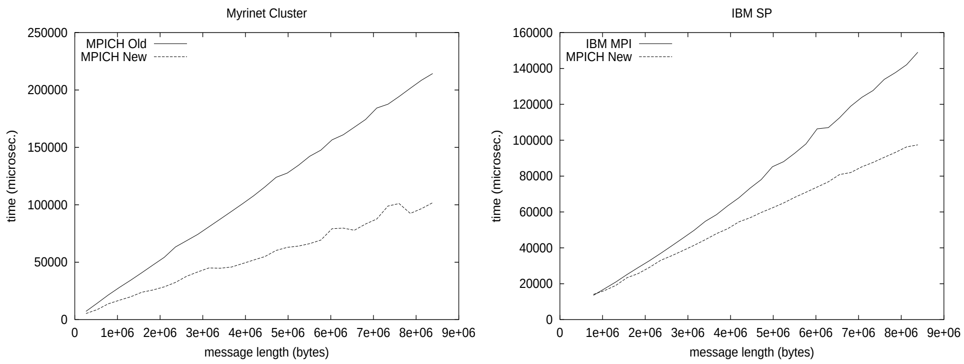

Fig. 4. Performance of long-message broadcast (64 nodes)

4.3 Reduce-Scatter

Reduce-scatter is a variant of reduce in which the result, instead of being stored at the root, is scattered among all processes. The old algorithm in MPICH implements reduce-scatter by doing a binary tree reduce to rank 0 followed by a linear scatterv. This algorithm takes $\lg p + p - 1$ steps, and the bandwidth term is $(\lg p + \frac{p-1}{p})n\beta$. Therefore, the time taken by this algorithm is $T_{old} = (\lg p + p - 1)\alpha + (\lg p + \frac{p-1}{p})n\beta + n \lg p\, \gamma$.

In our new implementation of reduce-scatter, for short messages, we use different algorithms depending on whether the reduction operation is commutative or noncommutative. The commutative case occurs most commonly because all the predefined reduction operations in MPI (such as `MPI_SUM`, `MPI_MAX`) are commutative.

For commutative operations, we use a recursive-halving algorithm, which is analogous to the recursive-doubling algorithm used for allgather (see Figure 5). In the first step, each process exchanges data with a process that is a distance $\frac{p}{2}$ away: Each process sends the data needed by all processes in the other half, receives the data needed by all processes in its own half, and performs the reduction operation on the received data. The reduction can be done because the operation is commutative. In the second step, each process exchanges data with a process that is a distance $\frac{p}{4}$ away: Each process sends the data needed by all processes in the other half of the current subtree, receives the data needed by all processes in its own half of the current subtree, and performs the reduction on the received data. This procedure continues recursively, halving the data communicated at each step, for a total of $\lg p$ steps. Therefore, if p is a power of two, the time taken by this algorithm is $T_{rec\_half} = \lg p\, \alpha + \frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma$. We use this algorithm for messages up to 512 KB.

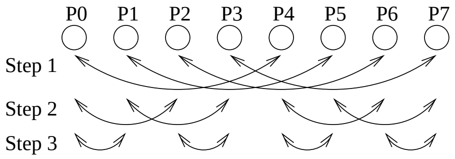

Fig. 5. Recursive halving for commutative reduce-scatter

If p is not a power of two, we first reduce the number of processes to the nearest lower power of two by having the first few even-numbered processes send their data to the neighboring odd-numbered process (rank+1). These odd-numbered processes do a reduce on the received data, compute the result for themselves and their left neighbor during the recursive halving algorithm, and, at the end, send the result back to the left neighbor. Therefore, if p is not a power of two, the time taken by the algorithm is $T_{rec\_half} = (\lfloor \lg p \rfloor + 2)\alpha + 2n\beta + n(1 + \frac{p-1}{p})\gamma$. This cost is approximate because some imbalance exists in the amount of work each process does, since some processes do the work of their neighbors as well. A similar method for handling non-power-of-two cases is used in [11].

If the reduction operation is not commutative, recursive halving will not work. Instead, we use a recursive-doubling algorithm similar to the one in allgather. In the first step, pairs of neighboring processes exchange data; in the second step, pairs of processes at distance 2 apart exchange data; in the third step, processes at distance 4 apart exchange data; and so forth. However, more data is communicated than in allgather. In step 1, processes exchange all the data except the data needed for their own result $(n - \frac{n}{p})$; in step 2, processes exchange all data except the data needed by themselves and by the processes they communicated with in the previous step $(n - \frac{2n}{p})$; in step 3, it is $(n - \frac{4n}{p})$; and so forth. Therefore, the time taken by this algorithm is $T_{short} = \lg p\alpha + n(\lg p - \frac{p-1}{p})\beta + n(\lg p - \frac{p-1}{p})\gamma$. We use this algorithm for very short messages ($<$ 512 bytes).

For long messages ($\geq$ 512KB in the case of commutative operations and $\geq$ 512 bytes in the case of noncommutative operations), we use a pairwise exchange algorithm that takes $p-1$ steps. At step i, each process sends data to $rank + i$, receives data from $rank - i$, and performs the local reduction. The data exchanged is only the data needed for the scattered result on the process ($\approx \frac{n}{p}$). The time taken by this algorithm is $T_{long} = (p-1)\alpha + \frac{p-1}{p}n\beta + \frac{p-1}{p}n\gamma$. Note that this algorithm has the same bandwidth requirement as the recursive halving algorithm. Nonetheless, we use this algorithm for long messages because it performs much better than recursive halving (similar to the results for recursive doubling versus ring algorithm for long-message allgather).

The SKaMPI benchmark, by default, uses a noncommutative user-defined reduction operation. Since commutative operations are more commonly used, we modified the benchmark to use a commutative operation, namely, `MPI_SUM`. Figure 6 shows the performance of the new algorithm for short messages on the IBM SP. The performance is significantly better than that of the algorithm used in IBM's MPI. For large messages, the new algorithm performs about the same as the one used in IBM's MPI. Because of a known problem in our Myrinet network, we were not able to test the old algorithm for short messages (the program hangs). We were able to test it for long messages on 32 nodes, and the results are shown in Figure 6. The new algorithm performs several times better than the old algorithm (reduce + scatterv) in MPICH.

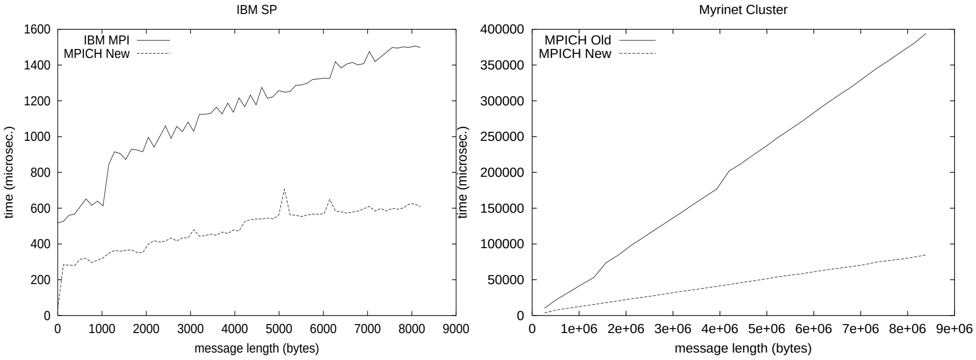

Fig. 6. Performance of reduce-scatter for short messages on the IBM SP (64 nodes) and for long messages on the Myrinet cluster (32 nodes)

4.4 Reduce

`MPI_Reduce` performs a global reduction operation and returns the result to the specified root. The old algorithm in MPICH uses a binary tree, which takes $\lg p$ steps, and the data communicated at each step is n. Therefore, the time taken by this algorithm is $T_{tree} = \lceil \lg p \rceil (\alpha + n\beta + n\gamma)$. This is a good algorithm for short messages because of the $\lg p$ steps, but a better algorithm, proposed by Rolf Rabenseifner [11], exists for long messages. The principle behind Rabenseifner's algorithm is similar to that behind Van de Geijn's algorithm for long-message broadcast. Van de Geijn implements the broadcast as a scatter followed by an allgather, which reduces the $n \lg p \beta$ bandwidth term in the binary tree algorithm to a $2n\beta$ term. Rabenseifner implements a long-message reduce effectively as a reduce-scatter followed by a gather to the root, which has the same effect of reducing the bandwidth term from $n \lg p\, \beta$ to $2n\beta$.

In the case of predefined reduction operations, we use Rabenseifner's algorithm for long messages ($>$ 2 KB) and the binary tree algorithm for short messages ($\leq$ 2 KB). In the case of user-defined reduction operations, we use the binary tree algorithm for all message sizes because in this case the user may pass derived datatypes, and breaking up derived datatypes to do the reduce-scatter is tricky. With predefined reduction operations, only basic datatypes are allowed.

The time taken by Rabenseifner's algorithm is the sum of the times taken by reduce-scatter (recursive halving) and gather (binary tree), which is $T_{rabenseifner} = 2 \lg p\, \alpha + 2\frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma$.

Figure 7 shows the performance of reduce for long messages on the Myrinet cluster. The new algorithm is more than twice as fast as the old algorithm in some cases. On the IBM SP we found that the new algorithm performs about the same as the one in IBM's MPI.

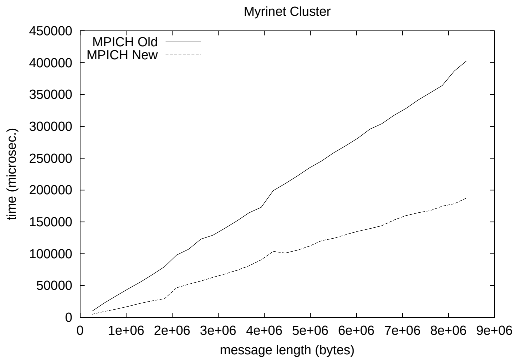

Fig. 7. Performance of reduce (64 nodes)

5 Conclusions and Future Work

We have reported on our work on improving the performance of the collective communication algorithms in MPICH. All these algorithms will be available in the next release of MPICH (1.2.6). Since these algorithms distinguish between short and long messages, an important factor is the message size at which we switch between the short- and long-message algorithms. At present, we use experimentally determined cut-off points, which are different for different algorithms. We plan to develop a model to calculate the cutoff points automatically based on system parameters. We also plan to extend this work to incorporate topology awareness, particularly algorithms that are optimized for architectures comprising clusters of SMPs. Finally, we plan to explore the use of one-sided communication to improve performance of collective operations.

Acknowledgments

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-ENG-38. We thank Rolf Rabenseifner for his careful reading and detailed comments on the paper.

References

1. M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor collective communication library (InterCom). In *Proceedings of Supercomputing '94*, November 1994.
2. M. Barnett, R. Littlefield, D. Payne, and R. van de Geijn. Global combine on mesh architectures with wormhole routing. In *Proceedings of the* 7^{th} *International Parallel Processing Symposium*, April 1993.
3. S. Bokhari. Complete exchange on the iPSC/860. Technical Report 91–4, ICASE, NASA Langley Research Center, 1991.
4. S. Bokhari and H. Berryman. Complete exchange on a circuit switched mesh. In *Proceedings of the Scalable High Performance Computing Conference*, pages 300–306, 1992.

5. Debra Hensgen, Raphael Finkel, and Udi Manbet. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.
6. L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A framework for collective personalized communication. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, 2003.
7. N. Karonis, B. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings of the Fourteenth International Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 377–384, 2000.
8. T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 131–140. ACM, May 1999.
9. P. Mitra, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Fast collective communication libraries, please. In *Proceedings of the Intel Supercomputing Users' Group Meeting*, June 1995.
10. Rolf Rabenseifner. Effective bandwidth (b_eff) benchmark. `http://www.hlrs.de/mpi/b_eff`.
11. Rolf Rabenseifner. New optimized MPI reduce algorithm. `http://www.hlrs.de/organization/par/services/models/mpi/myreduce.html`.
12. Peter Sanders and Jesper Larsson Träff. The hierarchical factor algorithm for all-to-all communication. In B. Monien and R. Feldman, editors, *Euro-Par 2002 Parallel Processing*, pages 799–803. Lecture Notes in Computer Science 2400, Springer, August 2002.
13. D. Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *Proceedings of the* 6^{th} *Distributed Memory Computing Conference*, pages 398–403, 1991.
14. Mohak Shroff and Robert A. van de Geijn. CollMark: MPI collective communication benchmark. Technical report, Dept. of Computer Sciences, University of Texas at Austin, December 1999.
15. Steve Sistare, Rolf vandeVaart, and Eugene Loh. Optimization of MPI collectives on clusters of large-scale SMPs. In *Proceedings of SC99: High Performance Networking and Computing*, November 1999.
16. V. Tipparaju, J. Nieplocha, and D.K. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS '03)*, 2003.
17. Jesper Larsson Träff. Improved MPI all-to-all communication on a Giganet SMP cluster. In Dieter Kranzlmuller, Peter Kacsuk, Jack Dongarra, and Jens Volkert, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting*, pages 392–400. Lecture Notes in Computer Science 2474, Springer, September 2002.
18. Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. In *Proceedings of SC99: High Performance Networking and Computing*, November 1999.
19. Thomas Worsch, Ralf Reussner, and Werner Augustin. On benchmarking collective MPI operations. In Dieter Kranzlmüller, Peter Kacsuk, Jack Dongarra, and Jens Volkert, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, pages 271–279. Springer, September 2002.