A Simple, Pipelined Algorithm for Large, Irregular All-gather Problems[⋆]

Jesper Larsson Träff[1], Andreas Ripke[1], Christian Siebert[1],
Pavan Balaji[2], Rajeev Thakur[2], and William Gropp[3]

[1] NEC Laboratories Europe, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
[2] Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439, USA
[3] Department of Computer Science
University of Illinois, Urbana, IL 61801, USA

Abstract. We present and evaluate a new, simple, pipelined algorithm for *large, irregular* all-gather problems, useful for the implementation of the `MPI_Allgatherv` collective operation of MPI. The algorithm can be viewed as an adaptation of a linear ring algorithm for regular all-gather problems for single-ported, clustered multiprocessors to the irregular problem. Compared to the standard ring algorithm, whose performance is dominated by the largest data size broadcast by a process (times the number of processes), the performance of the new algorithm depends only on the total amount of data over all processes. The new algorithm has been implemented within different MPI libraries. Benchmark results on NEC SX-8, Linux clusters with InfiniBand and Gigabit Ethernet, Blue Gene/P, and SiCortex systems show huge performance gains in accordance with the expected behavior.

1 Introduction

The *all-gather* problem is a basic collective communication operation, in which *each* participant of a predefined group wants to broadcast personal data to all other group members. In the MPI standard, this functionality is embodied in the *regular* `MPI_Allgather` collective, in which each process contributes the same amount of data, and in the *irregular* `MPI_Allgatherv` collective, where the amount of data can be freely chosen for the different processes [8]. For both MPI collectives, all participating processes know the sizes of the data to be broadcast by all other processes. The irregular all-gather operation is used for instance in linear algebra kernels for matrix multiplication and LU factorization [1].

The regular all-gather problem has been intensively studied (theoretically under the term *gossiping*, but is also known as *broadcast-to-all*, *all-to-all-broadcast*,

[⋆] This work was supported in part 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 DE-AC02-06CH11357.

as well as many other names) [5, 6], and many algorithms have been proposed and/or implemented as part of MPI libraries for various systems and communication models [1–3, 7, 9, 10]. The more challenging, irregular all-gather problem has received much less attention, and MPI libraries typically use the same algorithm for both `MPI_Allgather` and `MPI_Allgatherv`. For irregular problems with considerable differences between the amount of data contributed by the processes, this can have huge performance drawbacks. For extreme cases, the resulting performance loss can amount to orders of magnitude (cf. Section 3).

In this paper, we present an algorithm for large, irregular all-gather problems. The underlying idea is quite simple and can be viewed as an adaptation to the irregular problem of a ring-based algorithm for regular all-gather problems for single-ported, clustered multiprocessors. The algorithm has been implemented for several MPI libraries, and evaluated on diverse systems, namely NEC SX-8, two Linux clusters, IBM Blue Gene/P, and SiCortex 5832. We demonstrate significant performance improvements over a standard `MPI_Allgatherv` algorithm, depending on the amount of irregularity in the benchmark scenarios.

2 Algorithm and Implementation(s)

In the following, p is the number of participating (MPI) processes, numbered consecutively from 0 to $p-1$. We let m_i denote the size of the data contributed by process i, and $m = \sum_{i=0}^{p-1} m_i$ the total amount of data that eventually has to be gathered by all processes. For large data, we assume that the time for transmitting a message of size m' is simply $O(m')$. For most of the following discussion, a detailed communication cost model is unnecessary.

2.1 Standard, linear ring Algorithm

A basic (folklore) algorithm for large, regular all-gather problems is the *linear ring*. The algorithm performs $p-1$ communication rounds. In each round process i sends (starting with its own data) an already known block of data of size m' to process $(i+1) \bmod p$ and receives an unknown block of data from process $(i-1) \bmod p$. For regular problems where all blocks are of the same size $m_i = m'$, the completion time of the ring algorithm is $O((p-1)m') = O(m-m')$. The number of communication start-ups (latency) scales linearly with p. This is unproblematic for large m', but for small problems, an algorithm with a logarithmic number of start-ups is clearly preferable [1, 3, 10]. The linear ring algorithm is straightforward to implement. For systems with single-ported, bidirectional communication capabilities (where each process can at the same time send data to another process and receive data from a possibly different process) it can use the system communication bandwidth to full capacity. For irregular all-gather problems, where the data sizes m_i can vary arbitrarily over the processes, the algorithm can however perform poorly. The running time is determined by the largest amount of data $m' = \max_{i=0}^{p-1} m_i$, which has to be sent along the ring in each round, and is therefore $O((p-1)m')$. In particular, $(p-1)m'$ can be much larger than the total amount of data m.

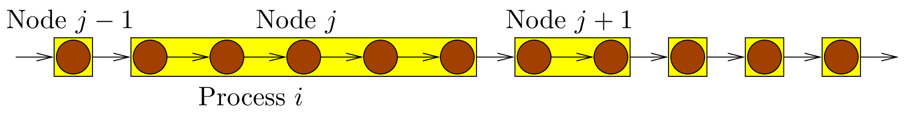

Fig. 1. The linear ring algorithm on a cluster of SMP nodes with different number of MPI processes per node. The processes are (virtually) ranked such that one process at each node receives data from another node, and one process sends data to another node in each round.

2.2 Pipelined (blocked) ring Algorithm

We first observe that the linear ring algorithm can also be used for the regular all-gather problems on clustered multiprocessors (like clusters of SMP nodes) with a single-ported communication network. In that case the ring is organized such that exactly one process i per SMP node has its predecessor $(i-1) \bmod p$ on another SMP node, and exactly one process j per SMP node has its successor $(j+1) \bmod p$ on another node. To accomplish this, a (virtual) reranking of the MPI processes might be necessary. The clustered, linear ring algorithm is now communication-bandwidth optimal, because in each round one process on each node receives a block of data and one process sends a block of data. This holds also for the case where the number of MPI processes per cluster node is not identical, and is illustrated in Figure 1.

In [11] it is observed that regular collective communication problems like the all-gather problem induce corresponding irregular problems over a set of nodes in a clustered system. Therefore, if the communication capabilities of processors and nodes in a cluster are similar (for instance, single ported), an algorithm for solving a regular problem on a clustered system (with possibly different number of processes per cluster node) can be used to solve its irregular counterpart over a set of processors. This observation can be exploited to convert the clustered linear ring algorithm into an algorithm for the irregular all-gather problem.

To accomplish this the data of process i of size m_i is associated with a virtual cluster node, and divided into $b_i = \max(1, \lceil m_i/B \rceil)$ blocks of size at most B. Each block is associated with a virtual processor in the node. The total number of blocks is $b = \sum_{i=0}^{p-1} b_i$ (note that $b \geq p$). Every actual process with data size m_i will play the role of a cluster node with b_i virtual processors. The linear ring algorithm with regular blocks of size (at most) B now solves the problem in $b-1$ instead of $p-1$ communication rounds. The resulting, *pipelined* (or *blocked*) *ring* algorithm is illustrated in Figure 2. Compared to the linear ring, the advantage of the pipelined ring algorithm is that (more) regular blocks are sent and received in each round, for a total time of $O((b-1)B)$. A small value for B increases the number of start-ups, and a large value increases the possible round up error. Therefore a proper balancing needs to be applied to find an optimal value for the block size parameter. We note that for extremely irregular all-gather problems where only one process has all the data, the pipelined ring algorithm is equivalent to a linear broadcast pipeline. For regular problems where $m_i = m'$ for all i, the

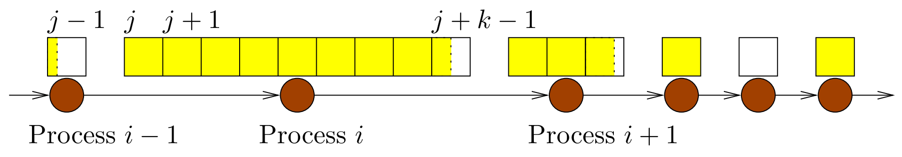

Fig. 2. The clustered, linear ring algorithm viewed as a pipelined (blocked) algorithm for solving the irregular all-gather problem. For each process, the data m_i is divided into blocks of some maximum block size B (partially full blocks are partially colored). Process i starts sending block $j+k-1$ and receiving block $j-1$. After $b-1$ rounds, where b represents the total number of blocks, all processes have gathered all the data.

block size B can be set to m', in which case the algorithm is identical to the standard, linear ring. Thus, by choosing B properly, the pipelined ring algorithm should never perform worse than the linear ring algorithm.

2.3 Determining an optimal block size

We note that for partially full blocks, only the actual data are sent and received (see again Figure 2). In particular, the empty blocks which arise for processes with $m_i = 0$ are neither sent nor received. Nevertheless, they contribute to the total number of communication rounds. We estimate the optimal block size B as follows, assuming that z denotes the number of processes with $m_i = 0$:

- If $z = 0$ we take $B = \min_{i=0}^{p-1} m_i$ (as long as this is not too small). This ensures that all processes are both sending and receiving blocks in (almost) all rounds.
- If $z \neq 0$ we try to minimize the time needed for $b-1$ communication rounds. Assuming that the remainders in the m_i/B terms are equally distributed, we get an average padding of $B/2$ for all partially full blocks. We can therefore simplify $b = \sum_{i=0}^{p-1} \max(1, \lceil m_i/B \rceil)$ to $b = \frac{m}{B} + \frac{p+z}{2}$. Assuming linear communication costs, where sending and receiving messages of size m' takes time $\alpha + \beta m'$, the estimated total running time is $(b-1)(\alpha + \beta B)$. Minimizing this term gives an (approximated) optimal block size of $B = \sqrt{\frac{2\alpha m}{\beta(p+z-2)}}$

3 Experimental Evaluation

We have benchmarked the new `MPI_Allgatherv` implementations with the following distributions of *contiguous data* over the p MPI processes. A base count c (which is varied over some interval) is used as seed for the following distributions:

1. **Regular:** all $m_i = c$ are identical, therefore $m = pc$.
2. **Broadcast:** $m_0 = c$, all other $m_i = 0$, therefore $m = c$.
3. **Spike:** similar to broadcast but all processes contribute some data, $m_0 = c/2$ and $m_i = c\frac{1}{2(p-1)}$, therefore $m = c$.

4. **Half full:** $m_{2\lfloor i/2\rfloor} = 2c$, and $m_{2\lfloor i/2\rfloor+1} = 0$, therefore $m = pc$.
5. **Linearly decreasing:** $m_i = 2c\frac{(p-1-i)}{p-1}$, therefore $m = pc$.
6. **Geometric curve:** $m_{i-1+j} = c\frac{p}{i\log p}$ for $i = 1, 2, 4, \ldots$ and $j = \{0, \ldots, i-1\}$, therefore $m = pc$.

In distributions (2) and (3) the same total amount of data $m = c$ is gathered by all processes, so similar running times can be expected (comparable to the regular distribution with p times smaller data size). The case for distributions (1), (4), (5) and (6) is analogous, where the total amount of data is $m = pc$.

We compare our implementations of the new `MPI_Allgatherv` algorithm with implementations of the standard linear ring algorithm that is still used in many MPI libraries [9]. The reported running times are minimum times for the last process to finish over a (small) number of iterations [4].

3.1 Results on an NEC SX-8 vector system

The pipelined ring has been implemented for MPI/SX for the NEC SX-series of parallel vector computers. It has been benchmarked with the distributions described above on 30 SX-8 nodes at HLRS in Stuttgart, with 1 and 8 MPI processes per node, respectively. Selected results are shown in Figure 3.

For the extreme broadcast distribution (2) the pipelined ring outperforms the standard linear ring by more than a factor of 10 on 30 SX-8 nodes. For 32 MBytes with a fixed block size B of 1 MByte an improvement of a factor $\frac{32\times 29}{29+31} \approx 15$ would have been best possible. Significant improvements can also be observed for the other distributions. The performance of the standard ring and the pipelined ring are similar for the regular (1) and the half full (4) distributions. Running on a randomly permuted communicator instead of `MPI_COMM_WORLD` gives almost identical results. This is a desirable property of an algorithm for a symmetric (i.e. non-rooted) collective operation like `MPI_Allgatherv` [12].

3.2 Results on a Linux Cluster with InfiniBand

To show the effect of the block size B, the algorithm has also been integrated into NEC's MPI/PC version and evaluated on an Intel Xeon based SMP cluster with InfiniBand interconnect. The running time is compared to the standard, non-pipelined algorithm for $B = 32K, 64K, 128K, 512K, 1024K$. Results are shown in Figure 4. For the spike distribution (3) the pipelined algorithm is faster for all block sizes. However, the best block size depends not only on the size of the problem but also on the distribution of data over the processes. This can be seen in the case of the decreasing distribution (5) where a too small block size makes the pipelined algorithm perform worse than the standard ring.

3.3 Results on a Linux Cluster with Gigabit Ethernet

We ran the benchmarks on a Linux cluster at Argonne National Laboratory with 24 nodes, each with two dual-core 2.8 GHz AMD Opteron CPUs (total

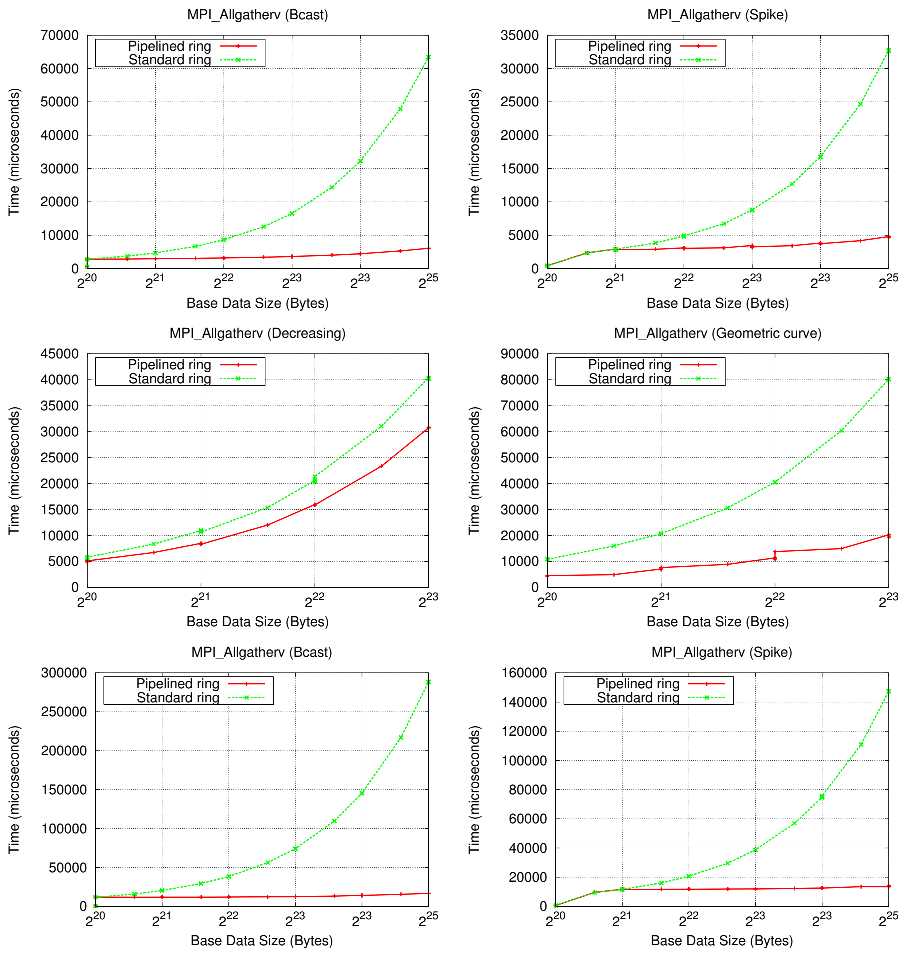

Fig. 3. Results (left to right, top to bottom) for distributions (2), (3), (5) and (6) on an NEC SX-8 with 30 nodes and 1 MPI process per node, and distributions (2) and (3) with 8 MPI processes per node. A fixed block size $B = 1$ MByte has been used. The base data size is the base count c multiplied by the size of an `MPI_INT`.

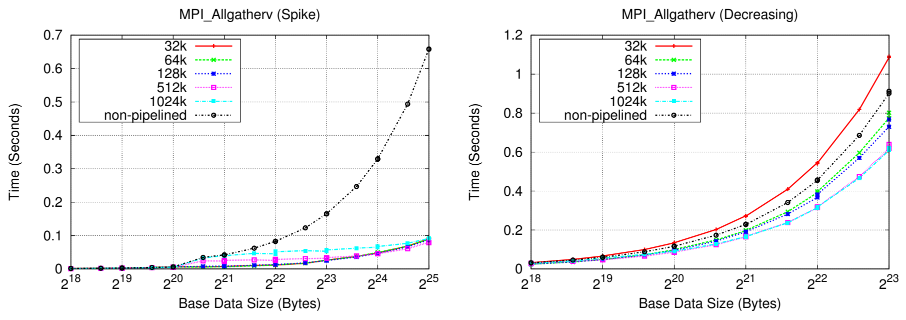

Fig. 4. Results from a Linux Xeon/InfiniBand cluster with 16 × 2 processes with spike (left) and linearly decreasing (right) distributions, and block size $B = 32K, 64K, 128K, 512K, 1024K$ compared to the non-pipelined algorithm.

of 4 cores per node or 96 cores in the system), and Gigabit Ethernet. We used MPICH2 1.0.7 as the MPI implementation. Selected results are shown in Figures 5 and 6. For small problem sizes, the pipelined algorithm performs only slightly better than the standard algorithm, but as problem size increases, the difference in performance becomes considerable. Figure 6(right) shows the distribution of communication and idle times for the two algorithms. As expected, the standard algorithm suffers because many processes remain idle for a long time, whereas in the pipelined algorithm, communication is more balanced. We also collected traces of the program execution and plotted them using the Jumpshot tool, as shown in Figure 7. The penalty due to idle time incurred by the standard algorithm is clearly visible as the yellow bars.

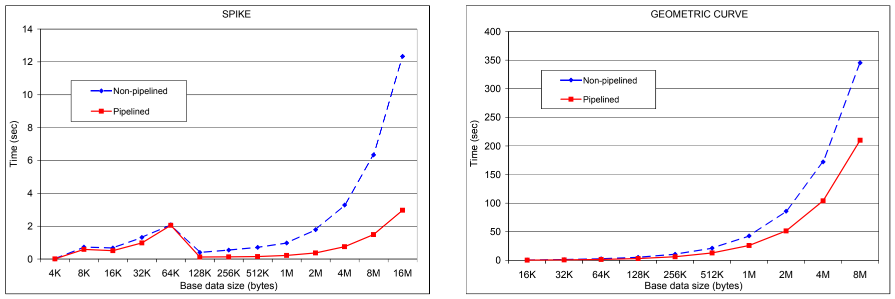

Fig. 5. Results with 96 processes on Linux cluster: (left) Spike distribution (right) Geometric curve distribution. A fixed block size B = 32 KB has been used.

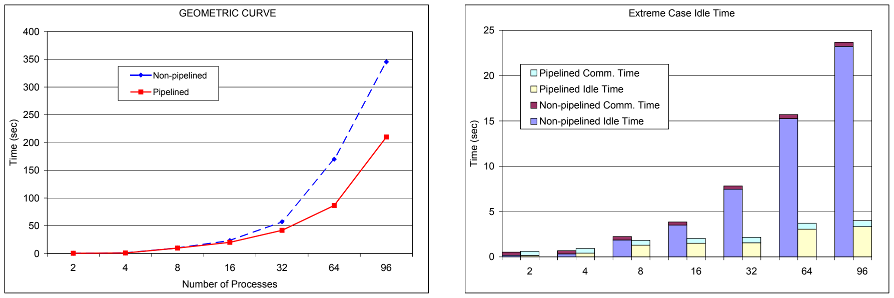

Fig. 6. Linux cluster: (left) Geometric curve distribution with varying number of processes, (right) Communication versus idle time in the extreme case of broadcast distribution

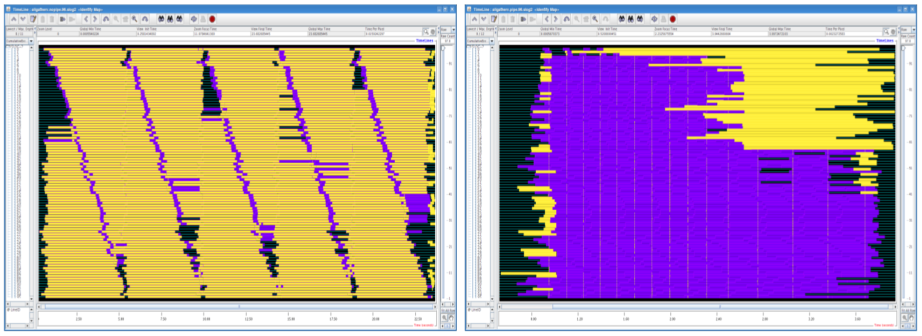

Fig. 7. Jumpshot plot of program trace on Linux cluster for several iterations of allgatherv with broadcast distribution: (left) Non-pipelined algorithm, (right) Pipelined algorithm. Yellow (light) is idle time, purple (dark) is communication time.

3.4 Results on SiCortex

Benchmarks were also performed on the SiCortex 5832 system at Argonne. This machine has 972 nodes, each with 6 cores, for a total of 5832 processors. The nodes are connected by a Kautz graph network. In some of our experiments the native SiCortex MPI implementation failed. We therefore implemented the standard linear ring algorithm ourselves and compared it with the pipelined algorithm. Figure 8 shows the results for a test run with a geometric curve dis-

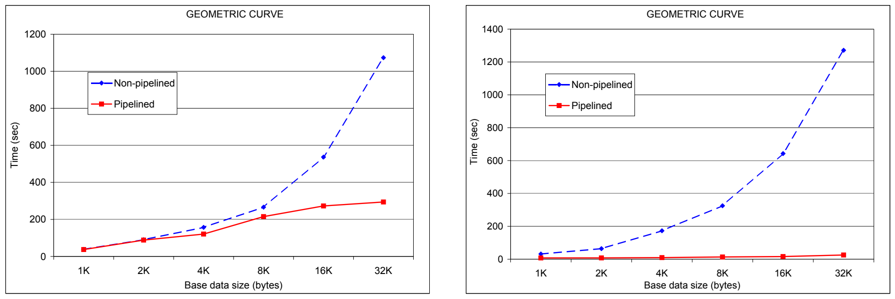

Fig. 8. Results for the geometric curve distribution: (left) with 5784 processes on the SiCortex machine and a fixed block size of $B = 1$ MB, (right) with 4096 processes on 1 rack of the Blue Gene/P and a fixed block size of $B = 64$ KB

tribution on 5784 processors. The pipelined algorithm significantly outperforms the standard algorithm as the message size increases.

3.5 Results on IBM Blue Gene/P

Finally, we performed the tests on one rack of the IBM Blue Gene/P at Argonne National Laboratory (4096 cores). The native implementation of `MPI_Allgatherv` in the Blue Gene/P's MPI library uses a very fast hardware-supported algorithm, which outperforms both standard ring and pipelined ring implementations. Therefore, to fairly compare pipelined and non-pipelined algorithms, we implemented both these algorithms. Figure 8 shows the results. The pipelined algorithm performs even better on this machine.

4 Concluding Remarks

We described a simple, pipelined ring algorithm for large, irregular all-gather problems. The algorithm was implemented within different MPI libraries and benchmarked on various systems, and in all cases showed considerable improvements over a commonly used linear ring algorithm for problems with significant irregularity in the individual message sizes. Determining the best possible pipeline block size for all distributions of input data still requires more (experimental) work. On regular problem instances the pipelined algorithm performs similarly to the linear ring, which is bandwidth optimal for that case. Ring algorithms can likewise be implemented to be largely independent on process placement in an SMP system. This is an important property for users expecting (self-) consistent performance of their MPI library [12].

References

1. P. Balaji, D. Buntinas, S. Balay, B. F. Smith, R. Thakur, and W. Gropp. Nonuniformly communicating noncontiguous data: A case study with PETSc and MPI. In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pages 1–10, 2007.
2. G. D. Benson, C.-W. Chu, Q. Huang, and S. G. Caglar. A comparison of MPICH allgather algorithms on switched networks. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 10th European PVM/MPI Users' Group Meeting*, volume 2840 of *Lecture Notes in Computer Science*, pages 335–343, 2003.
3. J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.
4. W. Gropp and E. 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*, volume 1697 of *Lecture Notes in Computer Science*, pages 11–18, 1999.
5. S. M. Hedetniemi, T. Hedetniemi, and A. L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.
6. D. W. Krumme, G. Cybenko, and K. N. Venkataraman. Gossiping in minimal time. *SIAM Journal on Computing*, 21(1):111–139, 1992.
7. A. R. Mamidala, A. Vishnu, and D. K. Panda. Efficient shared memory and RDMA based design for mpi_allgather over InfiniBand. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 66–75, 2006.
8. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.
9. R. Thakur, W. D. Gropp, and R. Rabenseifner. Improving the performance of collective operations in MPICH. *International Journal on High Performance Computing Applications*, 19:49–66, 2004.
10. J. L. Träff. Efficient allgather for regular SMP-clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 13th European PVM/MPI Users' Group Meeting*, volume 4192 of *Lecture Notes in Computer Science*, pages 58–65. Springer-Verlag, 2006.
11. J. L. Träff. Relationships between regular and irregular collective communication operations on clustered multiprocessors. Submitted, 2008.
12. J. L. Träff, W. Gropp, and R. Thakur. Self-consistent MPI performance requirements. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 14th European PVM/MPI Users' Group Meeting*, volume 4757 of *Lecture Notes in Computer Science*, pages 36–45. Springer-Verlag, 2007.