MPI on a Million Processors

Pavan Balaji[1], Darius Buntinas[1], David Goodell[1], William Gropp[2], Sameer Kumar[3], Ewing Lusk[1], Rajeev Thakur[1], and Jesper Larsson Träff[4]

[1] Argonne National Laboratory, Argonne, IL 60439, USA
[2] University of Illinois, Urbana, IL, 61801, USA
[3] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
[4] NEC Laboratories Europe, Sankt Augustin, Germany

Abstract. Petascale machines with close to a million processors will soon be available. Although MPI is the dominant programming model today, some researchers and users wonder (and perhaps even doubt) whether MPI will scale to such large processor counts. In this paper, we examine this issue of how scalable is MPI. We first examine the MPI specification itself and discuss areas with scalability concerns and how they can be overcome. We then investigate issues that an MPI implementation must address to be scalable. We ran some experiments to measure MPI memory consumption at scale on up to 131,072 processes or 80% of the IBM Blue Gene/P system at Argonne National Laboratory. Based on the results, we tuned the MPI implementation to reduce its memory footprint. We also discuss issues in application algorithmic scalability to large process counts and features of MPI that enable the use of other techniques to overcome scalability limitations in applications.

1 Introduction

We are fast approaching an era where the largest supercomputers in the world will have on the order of a million processor cores. MPI is the predominant model for programming the largest parallel machines today. As these machines scale to a million cores, many users and researchers are wondering whether MPI (and applications written in MPI) will scale to that level. In reality, there are multiple aspects to the scalability issue. First, is the MPI *specification* scalable, or are there aspects of the interface that may have issues at large scale? Second, is the MPI *implementation* scalable, and what do implementations need to address to improve their scalability? Third, are the parallel algorithms that MPI applications use themselves scalable to a million processes? We examine these issues in this paper.

Factors affecting scalability include performance and memory consumption. A *nonscalable* MPI function is one whose time or memory consumption *per process* increases linearly (or worse) with the number of processes, all other things being equal. For example, if the time taken by `MPI_Comm_spawn` increases linearly with the number of processes being spawned, it indicates a nonscalable implementation of the function. Similarly, if the memory consumption of

MPI_Comm_dup increases linearly with the number of processes, it is not scalable. Such examples of nonscalability need to be identified and fixed, both in the MPI specification and in implementations. A goal should be to use constructs that require only constant space per process.

2 Scalability Issues in the MPI Specification

Although the developers of MPI may not have envisioned million-core systems when MPI was first designed, it was nonetheless intended and designed with scalability in mind. For example, MPI tries to maintain very little global state per process. MPI also defines many operations as collective (called by a group of processes), which enables them to be implemented scalably and efficiently. Nonetheless, examination of the MPI specification reveals that some parts of it may have issues at large scale, particularly with respect to memory consumption.

2.1 Irregular Collectives

Many collectives in MPI have an irregular (or "v") version that allows users to transfer unequal amounts of data among processes. These collectives take one or more arguments that are arrays of size equal to the number of processes in the communicator, e.g., arrays of counts and displacements in MPI_Gatherv and MPI_Scatterv. An extreme case is MPI_Alltoallw, which takes *six* such arrays as arguments: counts, displacements, and datatypes for both sends and receives. Using such parameters is nonscalable: on a million processes, each array will consume 4 MB on each process.

Irregular collectives are often used in applications because MPI lacks any other way to express communication within a sparse subset of processes in a communicator. For example, in many applications that require nearest-neighbor communication in a Cartesian grid, each process performs an MPI_Alltoallv on MPI_COMM_WORLD and specifies 0 bytes for all processes other than its neighbors.[5] The PETSc library [13], for example, uses MPI_Alltoallv in this manner. While most MPI implementations optimize this pattern by communicating only with processes that have non-zero data, the MPI implementation must still scan through the entire array of data sizes to know which processes have non-zero data, and the user must allocate and initialize this array. On large numbers of processes, the time to read the entire array itself can be large and may increase linearly with system size, even though the number of neighbors a process communicates with remains fixed. Figure 1 shows this effect on an IBM Blue Gene/P for calling MPI_Alltoallv with zero-byte messages (no actual communication).

To avoid this problem, some computational libraries, such as PETSc, disable MPI_Alltoallv-based communication by default and instead perform direct

[5] This communication cannot be done easily by using subcommunicators because each process may belong to many subcommunicators and the collectives would have to be carefully ordered to avoid deadlocks. Such a scheme would also serialize much of the communication.

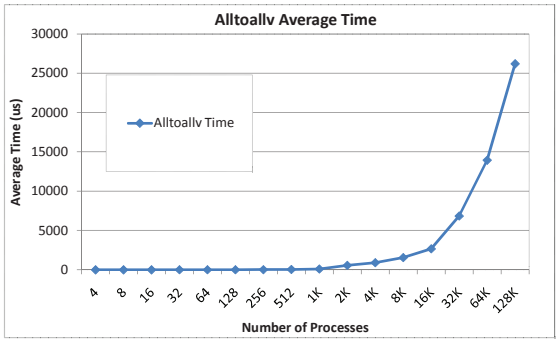

Fig. 1. Zero-byte Alltoallv time on IBM Blue Gene/P (no actual communication).

point-to-point communication among nearest neighbors, which is not as efficient as a concisely represented collective operation could be. The MPI Forum is working on fixing this issue in MPI-3. A proposal for sparse collectives has already been put forth [8].

2.2 Graph Topology

One of the most nonscalable constructs in MPI (memory wise) is the general graph topology. An MPI program can specify the communication pattern in the program as a graph, with edges of the graph representing the communication between nodes. This allows the MPI implementation to optimize communication by appropriate reordering or placement of processes. The problem with the specification is that it requires the *entire* communication graph to be supplied on each process. It therefore requires $O(p + e)$ space per process where e is the number of edges in the graph, and $O(p^2 + pe)$ in total (across all processes). Other limitations of this interface are discussed in [20].

The MPI Forum is currently discussing alternatives in which the graph is specified in a distributed form, requiring only $O(d)$ space per process (where d is the average degree of the communication graph) and $O(p + e)$ space in total across all processes [15]. It is then up to the MPI implementation to work in a distributed fashion and not construct the whole graph for each process.

2.3 One-Sided Communication

Many applications have been shown to benefit from one-sided communication, where a process directly accesses the memory of another process instead of sending and receiving messages. For this reason, MPI-2 also defined an interface for one-sided communication that uses put, get, and accumulate calls and three different synchronization methods. This interface, however, has not been widely used for a number of reasons, the main being that its performance is often worse than regular point-to-point communication. The culprit is often the synchronization associated with one-sided communication. The fence synchronization is

collective across the entire communicator associated with the window object, even if only small subsets of processes communicate with each other. The post-start-complete-wait method synchronizes over a smaller set, but it takes MPI process groups as arguments, which are somewhat cumbersome to create. The third method, lock-unlock, does not need synchronization with the target, but the origin must lock-unlock each target separately and the target does not know when the one-sided operation has completed. Other issues with this interface are discussed in [3].

Solutions to this problem are being considered by the MPI Forum for inclusion in MPI-3 as part of a new and better RMA interface for MPI [12].

2.4 All-to-All Communication

All-to-all communication is not a scalable communication pattern. Each process has a unique data item to send to every other process, which leads to limited opportunities for optimization compared with other collectives. This is not really a problem with the MPI specification but is something applications should be aware of and avoid as far as possible. Avoiding the use of all-to-all may require new algorithms.

2.5 Representation of Process Ranks

One nonscalable aspect of MPI is the explicit representation of process ranks, such as in the group routines `MPI_Group_incl` and `MPI_Group_excl`. While concise representations of collections of processes are possible (for example, some group routines support ranges), the MPI specification encourages the sort of unstructured enumeration that is difficult to scale. Eliminating the explicit enumeration should be considered as an option for large scale.

2.6 Fault Tolerance

On systems with a million cores, the probability of failure or unrecoverable error in some part of the system becomes very high. As a result, greater resilience against failure is needed from all components of the software stack, from low-level system software to libraries and applications. The MPI specification already provides some support to enable users to write programs that are resilient to failure, given appropriate support from the implementation [7]. For example, when a process dies, an implementation, instead of aborting the whole job, can return an error to any other process that tries to communicate with the failed process. It is then up to the application to decide what to do at that point.

However, more support is need for true fault tolerance. For example, the current set of error classes and codes need to be extended to indicate process failure and other failure modes. The MPI Forum has a subgroup on fault tolerance that is actively working on adding fault-tolerance capabilities to MPI-3 [11]. A number of research efforts in fault-tolerant MPI implementation also exist [4, 6, 9].

3 MPI Implementation Scalability

In terms of scalability, MPI implementations must pay attention to two aspects as the number of processes is increased: memory consumption of any function and the performance of *all* collective functions (including functions such as `MPI_Init` and `MPI_Comm_split`).

3.1 Process Mappings

MPI communicators usually contain a mapping from MPI process ranks to processor id's. This mapping is usually implemented by an array of p entries (where p is the number of processes in a communicator) for direct, constant-time lookup, possibly with shortcuts for particular mappings. A number of other mappings are often maintained, for instance, to enable fast navigation within and across the nodes of an SMP cluster. Although convenient and very fast, this solution, which requires linear space per process per communicator and quadratic space over the system, is clearly not scalable. Instead, communicators with the same process-to-processor mapping must share mappings. For example, if a communicator is dup'ed with `MPI_Comm_dup`, the new communicator must share the mapping with the original communicator. (At least MPICH2 and vendor implementations derived from it do so.) However, for random communicators, such as those created with `MPI_Comm_split`, mappings cannot be shared.

A solution to this problem is needed. Many mappings can be represented easily by simple linear functions, $ia + b \bmod p$. The identity mapping is often all that is needed for `MPI_COMM_WORLD`. Such linear representations, when possible, can be easily detected and cover many common cases, e.g., subcommunicators that form a consecutive segment from `MPI_COMM_WORLD`. However, this simple mapping covers only a very small fraction of the order of $p!$ possible communicators. More systematic approaches to compact representations of permutations have recently been explored in [2].

3.2 Memory Overheads in Communicator Creation

We ran some experiments on the IBM Blue Gene/P at Argonne to measure the memory overheads of creating new communicators. Figure 2 shows the results of a simple experiment to determine, for different numbers of processes, how many communicators can be created by calling `MPI_Comm_dup` of `MPI_COMM_WORLD` in a loop until it fails. Note that the maximum number of communicators supported by the implementation by default is 8189 (independent of `MPI_Comm_dup`) because of a limit on the number of available context ids.

With the default settings, the number of new communicators that can be created drops sharply starting at about 2048 processes. For 128K processes, the number drops to as low as 264. Although the MPI implementation does not duplicate the process-to-processor mapping in `MPI_Comm_dup`, it allocates some memory for optimizing collective communication. For example, it allocates memory to store "metadata" (such as counts and offsets) needed to optimize

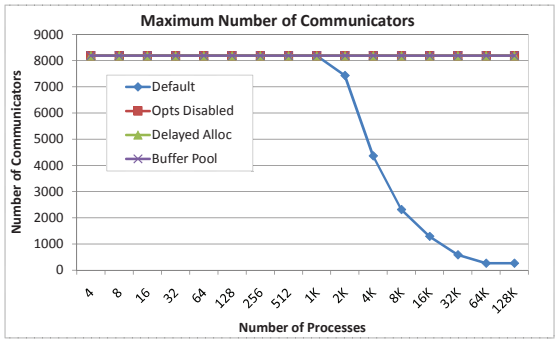

Fig. 2. Maximum number of communicators that can be created with `MPI_Comm_dup` of `MPI_COMM_WORLD` on IBM BG/P for different sizes of `MPI_COMM_WORLD`.

`MPI_Alltoall` and its variants. This memory is of size proportional to the number of processes in the communicator. Having such metadata per communicator is useful as it allows different threads to perform collective operations on different communicators in parallel. However, the per-communicator memory usage increases with system size. Since the amount of memory per process is very limited on the Blue Gene/P (512 MB), this optimization also limits the total number of communicators that can be created with `MPI_Comm_dup`.

This scalability problem can be avoided in a number of ways. The simplest way is to use an environment variable to disable collective optimizations (`DCMF_COLLECTIVES=0`), which eliminates the extra memory allocation. However, it has the undesirable impact of reducing the performance of all collectives. Another approach is to use an environment variable (`DCMF_ALLTOALL_PREMALLOC=N`) that delays the allocation of memory until the user actually calls `MPI_Alltoall` on the communicator. This approach helps only those applications that do not perform `MPI_Alltoall`. A third approach that we have implemented is to allocate memory based on the amount of required parallelism rather than the number of communicators. Since the per-communicator metadata buffers are intended to enable different threads to perform collective operations on different communicators in parallel, a fixed number of buffers equal to the maximum number of threads allowed on a node of the Blue Gene/P (four) would be sufficient.

Figure 3 shows the memory consumption in all these cases after 32 calls to `MPI_Comm_dup`. The fixed buffer pool enables all optimizations for all collectives and takes up only a small amount of memory.

3.3 Scalability of `MPI_Init`

Since the performance of `MPI_Init` is rarely measured, implementations may neglect scalability issues in `MPI_Init`. On large numbers of processes, however, a nonscalable implementation of `MPI_Init` may result in `MPI_Init` itself taking several minutes. For example, on connection-oriented networks where a process

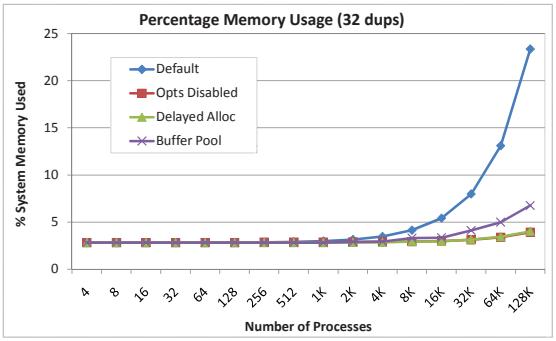

Fig. 3. MPI memory usage on BG/P (after 32 calls to `MPI_Comm_dup` of `MPI_COMM_WORLD`)

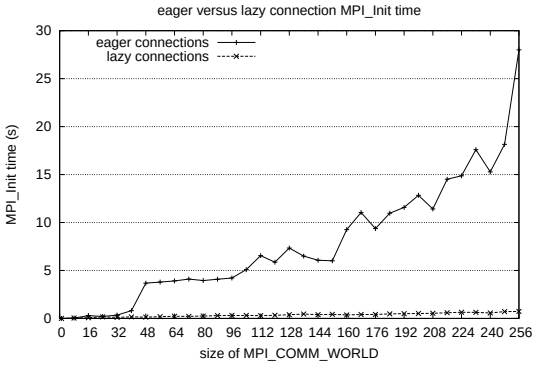

Fig. 4. Duration of `MPI_Init` with eager versus lazy connection establishment. Results are from an eight-core per node cluster using TCP/IP as the communication protocol.

needs to establish a connection with another process before communication, it is tempting for an MPI implementation to set up all-to-all connections in `MPI_Init` itself. This is however an $O(p^2)$ operation and hence inherently nonscalable. A better approach is to establish no connections in `MPI_Init` and instead establish a connection when a process needs to communicate with another. This method does make the first communication more expensive, but only those connections that are really needed are set up. It also minimizes the number of connections as applications written for scalability are not likely to have communication patterns where all processes directly communicate with all other processes.

Figure 4 shows the time taken by `MPI_Init` on a Linux cluster with TCP when all connections are set up eagerly in `MPI_Init` and when they are set up lazily. The eager method is clearly not scalable.

3.4 Scalable Algorithms for Collective Communication

MPI libraries typically use $O(\log p)$ algorithms (such as a binomial tree) for short-message collectives and $O(m)$ algorithms, where m is the total message size, for large-message collectives (e.g., a broadcast implemented as a scatter followed by an allgather) [19]. On a million-processor system, we can continue to use $O(\log p)$ algorithms for short messages. For large messages, however, an $O(m)$ broadcast algorithm may not scale, as the message size in the allgather phase will be very small. For example, for a 1 MB broadcast on one million processes, the allgather phase may involve one byte messages. Hybrid algorithms [21] can first do a logarithmic broadcast to a subset of nodes and then a scatter/allgather on many subsets at the same time. Since a million-processor system will likely have several large multicore nodes connected by a direct or switched network, a hybrid collective that first does a network broadcast on the different nodes followed by an intranode broadcast may be a more scalable solution depending on the scalability of the memory hierarchy within a multicore node.

Topology-specific optimizations may also be useful. For easy assembly, most interconnects have smaller diameters than the size of the network ($O(\log p)$ on switched networks and $O(\sqrt[3]{p})$ on 3D torus networks). A pipelined algorithm that streams data on a spanning tree embedded in the network topology will provide more scalable performance because the throughput of the collective is determined by $\frac{\text{message\_size}}{\text{diameter}}$. For example, on the BG/P, the six-color torus algorithm can keep 95% of all the links busy during an 8 MB broadcast operation [10].

Global collective acceleration supported by many networks such as Quadrics, InfiniBand, and Blue Gene may be another solution for collectives on `MPI_COMM_WORLD`. On the Blue Gene/P, for example, broadcast, reduce, allreduce, scatter, scatterv, and allgather collectives take advantage of the combine and broadcast features of the tree network.

4 Enabling Application Scalability

As emerging hardware architectures make greater degrees of parallelism available, even necessary, existing applications are facing the problem of scaling up. The complexity of solving this problem depends entirely on the basic algorithms used by the application, and so no completely general approach will do. In this section, we describe some ways in which features of MPI, perhaps not being used in the current version of a particular application, can play an important role in enabling that application to run effectively on more processors. In many cases, it may be possible to retain most of the existing application code, which is of course extremely desirable from the application's point of view.

4.1 Higher-Dimensional Decompositions with MPI

One relatively straightforward case occurs when the application consists of calculations carried out on a rectangular two- or three-dimensional mesh with nearest-neighbor communication, but the application has parallelized the computation

with a one-dimensional decomposition of the mesh. This approach results in contiguous buffers for the MPI sends and receives, which simplifies the application. Straightforward arithmetic shows that as the number of processors and mesh cells scales up, it becomes more efficient to use a two- or three-dimensional decomposition of the mesh. This results in noncontiguous communication buffers for sending and receiving edge or face data. MPI can help by providing the functions for assembling MPI datatypes that describe these noncontiguous areas of memory. Modern MPI implementations then use particularly efficient algorithms for communicating these areas [18].

4.2 Use of Threads with MPI

In the earlier parts of this paper, we have been treating "MPI on a million processors" as if it meant that the application would see one million separate MPI ranks. This is unlikely to be the case in practice. As the amount of memory per core decreases, applications will be increasingly motivated to use a shared-memory programming model on multicore nodes, while continuing to use MPI for communication among address spaces. MPI supports this transition by having clear semantics for interoperation with threads, based on four levels of thread safety that can be required by an application and provided by an MPI implementation. Although no particular thread system is mentioned in the MPI standard, the MPI specification of levels of thread safety meshes particularly well with the OpenMP standard. This feature has made OpenMP+MPI the current most widely used hybrid programming method [5, 16, 17].

4.3 Use of MPI-Based Libraries to Hide Complexity

We describe an example of how MPI enables the development of libraries that make it easier to write applications.

One of the most obviously nonscalable approaches to parallel programming is the "manager-worker" algorithm, which achieves good load balancing at the expense of having a single manager process to coordinate the dispensing of work to the worker processes, collection of results, and perhaps addition of new work to the work queue. We recently worked with a Monte Carlo application in nuclear physics [14], which used a variation of this approach and was stuck at about 2000 processors, with the ambition of going to tens of thousands. MPI helped solve this problem by enabling the construction of a general-purpose library called ADLB (Asynchronous Dynamic Load Balancing) [1] that eliminated the single manager as a bottleneck by providing a simple put/get interface to a distributed work queue. The application actually became simpler than before as the MPI communication disappeared; any application process simply puts new work to the queue or retrieves work from it. The ADLB implementation, however, is relatively complex and for scalability and efficiency requires a full range of MPI features, including thread safety, multiple communicators, derived datatypes, asynchronous sends and receives, the "ready" send operation, and

other features, all of which are hidden from the application. In this way, MPI supports application scalability while actually simplifying application code.

5 Conclusions

MPI is ready for scaling to a million processors barring a few issues that can be (and are being) fixed. Nonscalable parts of the MPI standard include irregular collectives and virtual graph topology. There is also a need for investigating systematic approaches to compact, adaptive representations of process groups. MPI implementations must pay careful attention to the memory requirements of functions and systematically root out data structures whose size grows linearly with the number of processes. To obtain scalable performance for collective communication, MPI implementations may need to become more topology aware or rely on global collective acceleration support. MPI also provides other features, such as support for building complex libraries and clear semantics for interoperation with threads, that enable applications to use other techniques to scale when limited by memory or data-size constraints.

Acknowledgments

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357 and award DE-FG02-08ER25835, and in part by the National Science Foundation award 0837719.

References

1. ADLB library. `http://www.cs.mtsu.edu/~rbutler/adlb/`.
2. Jérémy Barbay and Gonzalo Navarro. Compressed representations of permutations, and applications. In *Proc. of 26th Int'l Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 111–122, 2009.
3. Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In *2nd Workshop on Hardware/Software Support for High Performance Sci. and Eng. Computing*, 2003.
4. George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cedile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vencent Neri, and Anton Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Proc. of SC 2002*. IEEE, 2002.
5. Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, 2007.
6. Graham E. Fagg and Jack J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Proc. of Euro PVM/MPI 2000*, pages 346–353. Lecture Notes in Computer Science 1908, Springer, September 2000.
7. William D. Gropp and Ewing Lusk. Fault tolerance in MPI programs. *Int'l Journal of High Performance Computer Applications*, 18(3):363–372, 2004.

8. Torsten Hoefler and Jesper Larsson Träff. Sparse collective operations for MPI. In *Proc. of 14th Int'l Workshop on High-level Parallel Programming Models and Supportive Environments at IPDPS*, 2009.
9. Hideyuki Jitsumoto, Toshio Endo, and Satoshi Matsuoka. ABARIS: An adaptable fault detection/recovery component framework for MPIs. In *Proc. of 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS '07) in conjunction with IPDPS 2007*, March 2007.
10. Sameer Kumar, Gabor Dozsa, Jeremy Berg, Bob Cernohous, Douglas Miller, Joseph Ratterman, Brian Smith, and Philip Heidelberger. Architecture of the Component Collective Messaging Interface. In *Proc. of Euro PVM/MPI 2008*, pages 23–32, September 2008.
11. MPI Forum fault tolerance working group. `https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/FaultToleranceWikiPage`.
12. MPI Forum RMA working group. `https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/RmaWikiPage`.
13. PETSc library. `http://www.mcs.anl.gov/petsc`.
14. Steven C. Pieper and R. B. Wiringa. Quantum Monte Carlo Calculations of Light Nuclei. *Annu. Rev. Nucl. Part. Sci.*, 51:53, 2001.
15. Proposal for distributed graph topology. `https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/33`.
16. Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Proc. of 17th Euromicro Int'l Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009)*, pages 427–236, February 2009.
17. Ashay Rane and Dan Stanzione. Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems. In *Proc. of 10th LCI Int'l Conference on High-Performance Clustered Computing*, March 2009.
18. Robert Ross, Neill Miller, and William Gropp. Implementing fast and reusable datatype processing. In *Proc. of EuroPVM/MPI 2003*, pages 404–413, September 2003.
19. Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *Int'l Journal of High-Performance Computing Applications*, 19(1):49–66, Spring 2005.
20. Jesper Larsson Träff. SMP-aware message passing programming. In *Proc. of 8th Int'l Workshop on High-level Parallel Programming Models and Supportive Environments at IPDPS 2003*, pages 56–65, 2003.
21. Jesper Larsson Träff. A simple work-optimal broadcast algorithm for message-passing parallel systems. In *Proc. of Euro PVM/MPI 2004*, pages 173–180, September 2004.