Open Issues in MPI Implementation

Rajeev Thakur and William Gropp

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

Abstract. MPI (the Message Passing Interface) continues to be the dominant programming model for parallel machines of all sizes, from small Linux clusters to the largest parallel supercomputers such as IBM Blue Gene/L and Cray XT3. Although the MPI standard was released more than 10 years ago and a number of implementations of MPI are available from both vendors and research groups, MPI implementations still need improvement in many areas. In this paper, we discuss several such areas, including performance, scalability, fault tolerance, support for debugging and verification, topology awareness, collective communication, derived datatypes, and parallel I/O. We also present results from experiments with several MPI implementations (MPICH2, Open MPI, Sun, IBM) on a number of platforms (Linux clusters, Sun and IBM SMPs) that demonstrate the need for performance improvement in one-sided communication and support for multithreaded programs.

1 Introduction

MPI (the Message Passing Interface) is a widely used paradigm for parallel programming. It is used across the entire spectrum of parallel machines—from small Linux clusters to the largest parallel machines in the world such as IBM Blue Gene/L and Cray XT3. The MPI standard has existed for a long time—MPI-1 was released in 1994 and MPI-2 in 1997—and a number of MPI implementations are available. Free, portable implementations include MPICH, MPICH2, MVAPICH, MVAPICH2, LAM, and Open MPI. In addition, all computer-system and network-hardware vendors (such as IBM, Cray, Sun, HP, SGI, Intel, Microsoft, NEC, Hitachi, Fujitsu, Myricom, Quadrics, Mellanox, and QLogic) provide implementations of MPI. (Many of the vendor implementations are derived from the public-domain implementations.) Although MPI implementations have matured over the years, improvements are still needed in a number of areas. It is not sufficient just to provide the lowest possible ping-pong latency and highest possible large-message bandwidth between two processes. Users expect good performance across all aspects of the MPI standard.

In this paper, we discuss several areas in which MPI implementations still need improvement. These include performance, scalability, fault tolerance, support for debugging and verification, topology awareness, collective communication, derived datatypes, parallel I/O, one-sided communication, and support for

multithreaded programs. For the last two areas, we also present results from experiments with several MPI implementations (MPICH2, Open MPI, Sun, IBM) on a number of platforms (Linux clusters, Sun and IBM SMPs) that demonstrate the need for performance improvements.

2 Areas Needing Improvement in MPI Implementations

Below we discuss in broad terms several areas in which better support is needed from MPI implementations. It is not a comprehensive list, but it covers most of the important topics.

2.1 Basic Performance

The holy grail of message-passing performance is to achieve sub-microsecond latency for short messages. That goal has already been achieved on shared-memory machines [4] but not yet on distributed-memory systems. In addition to achieving low latency and high bandwidth on ping-pong benchmarks, it is essential to deliver good performance across the entire range of message sizes, avoiding sharp jumps in between. However, this is not the case in many MPI implementations that use a different protocol for short and long messages (eager versus rendezvous delivery) to minimize the need for internal buffering. An example in shown in Figure 1: On the IBM Blue Gene/L, a large jump occurs around 1024 bytes because of the transition from eager to rendezvous protocol. Smoothing out such performance jumps is a difficult challenge because of the tradeoffs between performance and resource consumption.

Another basic performance requirement is that a user should be able to achieve better (or equal) performance by using a single MPI function than by using a combination of other MPI functions that can implement the same functionality [29]. This requirement is not met in some cases. For example, in Figure 1, a user with a 1500-byte message will achieve better performance by sending two 750-byte messages. More such examples can be found in [29].

2.2 Scalability

MPI implementers must bear in mind that the number of processes in an MPI application may no longer be limited to a few hundred or a few thousand. Machines with much larger numbers of processors already exist. For example, the IBM Blue Gene/L at Lawrence Livermore National Laboratory has 131,072 processors. Larger systems are expected in the near future. As a result, MPI implementations must pay close attention to aspects of their code that grow linearly with the number of processors. Such aspects include the size of internal data structures, the number of connections established during `MPI_Init`, and the complexities of algorithms used anywhere in the implementation. Connecting all processes to each other in `MPI_Init` is no longer an option. If the underlying network requires connections, they must be set up only if and when

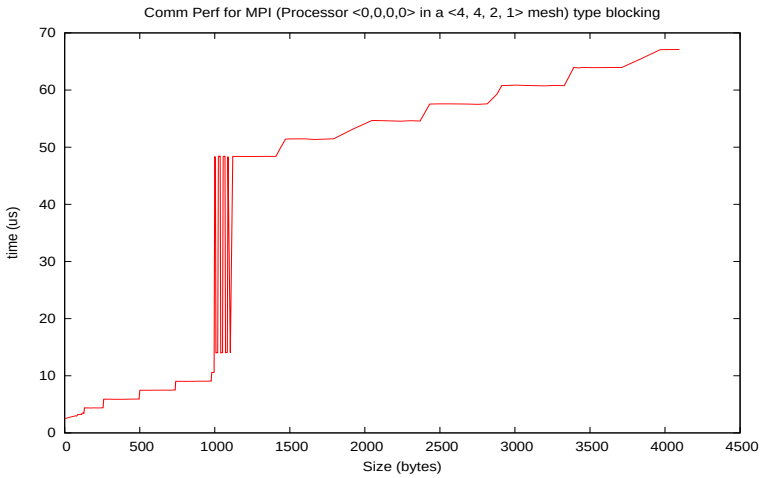

Fig. 1. Measured performance of short messages on IBM Blue Gene/L. Note the large jump around 1024 bytes; this is the transition from eager to rendezvous protocol in the MPI implementation.

needed, for example, when a process first tries to communicate with another process (MPICH2 already does this). Similarly, connections may need to be closed dynamically if they remain idle for a long time; and collective-communication algorithms, including all-to-all, may need to limit the number of connections.

2.3 Fault Tolerance

Closely tied to scalability is the need for increased tolerance to failure. As systems get larger, the probability of failure of components is larger. MPI implementations must improve their ability to handle failures, such as broken connections and dead processes, to the extent possible. A number of research efforts in fault-tolerant MPI implementation exist [2, 7, 13]. However, production MPI implementations need to improve their support for fault tolerance. In addition, fault tolerance often comes at a cost, and a careful balance must be struck between performance and fault tolerance.

2.4 Support for Debugging and Verification

MPI applications can be difficult to write and equally difficult to debug. To help application programmers, MPI implementations must provide better support for tools that help with debugging and verification. For example, MPI implementations must integrate better with parallel debuggers (e.g., TotalView) [6, 9]. Auxiliary tools that help in debugging are also useful. An example is the `collchk` library [8] provided with MPICH2 that can check for inconsistencies in

parameters passed to collective functions on different processes, such as the root for an `MPI_Bcast`. In a large and complex application with many MPI functions, `collchk` has helped find bugs that would otherwise have been very difficult to catch. A similar tool is described in [31]. Other tools also exist for checking program correctness, such as MARMOT [16], Umpire [32], and Intel Trace Analyzer and Collector [12], but more work is needed in this area.

Parallel programs are also prone to suffer from deadlocks and race conditions that may remain undetected for a long time because they are timing dependent [17]. For example, the byte-range locking algorithm proposed in [27] has a race condition that results in deadlock. It was discovered only a year later with the help of formal-verification methods [18]. Easy-to-use tools that use formal verification would be invaluable.

2.5 Virtual-to-Physical Topology Mapping

Today's large parallel machines, such as IBM Blue Gene/L and Cray XT3, have nodes arranged in a 3D torus topology. On such machines, it is more efficient to have MPI processes mapped on the nodes in a way that results in the majority of the communication taking place between nearest neighbors in the torus. MPI defines process-topology functions that allow users to create virtual process topologies and organize their communication among nearest neighbors on such topologies. However, the MPI implementation must efficiently map the virtual topology onto the physical processor layout such that nearest neighbors in the virtual topology are also nearest neighbors in the physical topology. This efficient mapping is often lacking in MPI implementations and must be provided. Applications may also need `MPI_COMM_WORLD` to be mapped appropriately on the machine.

2.6 Derived Datatypes

Derived datatypes in MPI allow users to specify noncontiguous memory layouts and thereby communicate noncontiguous data with a single function call. They are intended to provide higher performance than having the user pack all the data contiguously before calling MPI. However, MPI implementations have historically performed very poorly with derived datatypes, to the extent that users don't even think about using them. This situation defeats the purpose of having derived datatypes in the standard. Although some research efforts have optimized the processing of derived datatypes [21, 30], their performance still often lags behind that of manual packing. One promising effort demonstrated higher performance than user packing by exploiting knowledge of the memory architecture of the machine and doing memory copies efficiently [5]. However, this work is yet not incorporated in the official release of MPICH2. More research, development, and incorporation into widely used MPI implementations clearly is needed for derived datatypes.

2.7 Collective Communication

MPI collective communication functions, such as broadcast and reduce, play a big role in helping applications achieve good performance. Although a lot of research has been done on collective communication algorithms [1, 3, 28] and some implementations have incorporated optimized algorithms [19, 26], more work is still needed in some areas. For example, with the advent of multicore chips, MPI applications will routinely have multiple processes on a single node connected with multiple processes on other nodes by an interconnection network. Therefore, collective communication algorithms must be designed to effectively use such a hierarchical communication topology. Although research has been done on topology-aware collectives [14, 15, 22], not all production implementations have incorporated such algorithms yet. Furthermore, optimized algorithms are needed for the entire set of collectives in MPI, not just a select few.

The best algorithm for a particular collective communication function often depends on the message size and number of processes. In MPICH2, for example, the MPI collective functions use multiple algorithms, and one of them is selected for a specific message size and number of processes [26]. However, the cutoff points for switching between algorithms are based on measurements performed some time ago on one platform. They may not be right for other platforms. A better approach is needed that determines the right cutoff points for the specific machine being used. Dynamic tuning of algorithms may also be needed.

2.8 Parallel I/O

MPI-2 includes an interface for parallel file I/O, commonly referred to as MPI-IO. The most commonly used implementation of MPI-IO is ROMIO [20, 24]. To our knowledge, almost all MPI implementations, except IBM's MPI for the SP, use ROMIO as the basis for MPI-IO. Although ROMIO has many optimizations that improve I/O performance substantially, such as data sieving and collective I/O [25], more work is needed in improving those algorithms and selecting the right internal buffer sizes for I/O and the right number of I/O aggregators on large systems. Applications would also benefit from a production-quality client-side caching system that can take advantage of the default weak consistency semantics of MPI-IO. Furthermore, many implementations do not yet support the portable external32 data format and user-defined data representations that are part of the MPI standard. These features are needed for standards compliance and for being able to write files that can be read on any architecture.

In the following sections, we discuss in greater detail two other areas needing improvement, namely, one-sided communication and support for multithreaded programs.

3 One-Sided Communication

The MPI-2 standard added one-sided communication operations to MPI. These operations offer a different programming model from the regular MPI-1 point-

to-point operations: A process can directly write to or read from the memory of a remote process via *put* and *get* operations. A key feature of MPI one-sided communication is that data transfer and synchronization are separated. This feature allows multiple transfers to use a single synchronization operation, thus reducing the total overhead. MPI supports three synchronization methods: fence (collective synchronization), post-start-complete-wait (only communicating processes synchronize), and passive target (only the origin process calls lock-unlock functions).

To test how MPI implementations perform for one-sided communication, we wrote a benchmark that mimics the common "halo exchange" (or ghost-cell exchange) operation in applications that approximate the solution to partial differential equations. The code for this communication pattern, using MPI point-to-point communication, is as follows.

```
for (j=0; j<n_partners; j++) {
    MPI_Irecv( rbuffer[j], len, MPI_BYTE, partners[j], 0,
               MPI_COMM_WORLD, &req[j] );
    MPI_Isend( sbuffer[j], len, MPI_BYTE, partners[j], 0,
               MPI_COMM_WORLD, &req[n_partners+j] );
}
MPI_Waitall( 2*n_partners, req, MPI_STATUSES_IGNORE );
```

We wrote a number of versions of this benchmark with one-sided communication and using all three synchronization mechanisms. We ran the benchmark on a Sun Fire SMP at the University of Aachen and an IBM p655+ SMP at the San Diego Supercomputer Center using the native vendor MPI implementations (Sun and IBM).

Figure 2 shows a subset of the results on the Sun and IBM machines. The results on the Sun machine indicate that it is possible to get good performance with one-sided communication; in fact, on this system the performance with lock-unlock synchronization is better than with point-to-point communication. On the other hand, the IBM system performs very poorly for one-sided communication. With eight processes on an eight-node SMP, the one-sided communication performance was on the order of forty times slower than the point-to-point performance (data not shown). With seven processes on the same eight-node SMP, the one-sided communication performance is still poor (as shown in Figure 2) but an order of magnitude faster than with eight processes. The significant change in performance between eight and seven processes suggests that a thread is used for implementing the one-sided communication operations and that the implementation is not prepared to handle the case where there are more threads than processors. The results on the IBM machine demonstrate that efforts are needed to improve the performance of MPI one-sided communication.

Additional results for other MPI implementations and platforms can be found in [11].

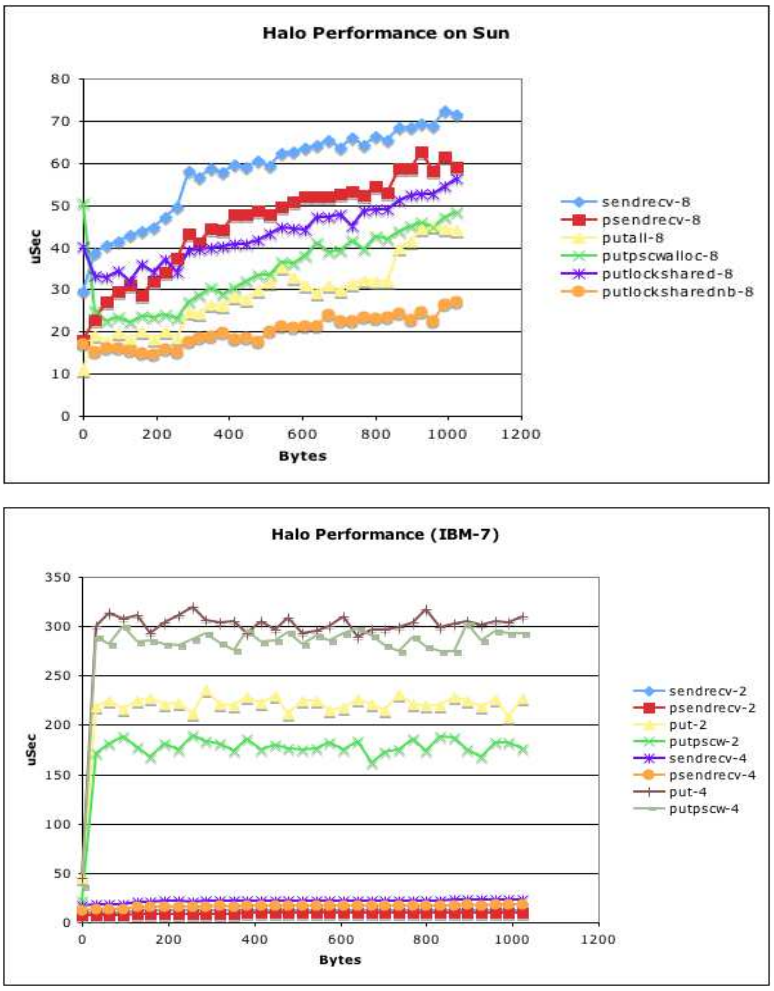

Fig. 2. Performance of one-sided communication for halo exchange on Sun Fire with 16 processes (top) and IBM p655+ with 7 processes (bottom). putall is the fence version with all assert options, putpscwalloc is the post-start-complete-wait synchronization with `MPI_Alloc_mem`, putlockshared is passive target with shared locks, and putlocksharednb omits the barrier that is necessary to ensure completion at the target.

4 Efficient Support for MPI_THREAD_MULTIPLE

MPI-2 allows users to write multithreaded MPI programs and defines the interaction between MPI and threads. MPI implementations that support the highest level of thread safety for user programs, `MPI_THREAD_MULTIPLE`, are becoming widely available. Thread safety does not come for free, however, because the implementation must protect certain data structures or parts of the code with mutexes or critical sections. Developing a thread-safe MPI implementation is a fairly complex task, and the implementers must make several design choices, both for correctness and for performance [10]. To simplify the task, implementations often focus on correctness first and performance later (if at all). As a result, even though an MPI implementation may support multithreading, its performance may be far from optimized.

To determine how current implementations perform, we ran tests that measure the bandwidth and latency obtained when multiple threads of a process communicate with multiple threads of another process compared with multiple processes instead of threads (see Figure 3). We ran the tests on the Sun Fire and IBM p655+ SMPs and on a Linux cluster at Argonne National Laboratory. The cluster has nodes with two dual-core AMD Opterons and Gigabit Ethernet as the interconnect. We used the native vendor MPI implementations on the Sun and IBM machines and two implementations on the Linux cluster: MPICH2 and Open MPI.

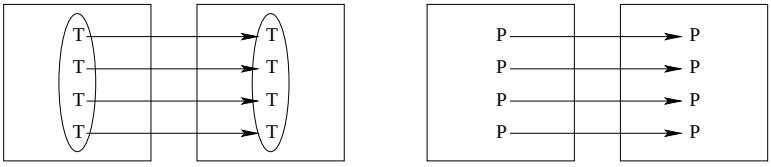

Fig. 3. Communication test when using multiple threads (left) versus multiple processes (right).

The first test measures the cumulative bandwidth obtained and demonstrates how much thread locks affect the cumulative bandwidth; ideally, the multiprocess and multithreaded cases should perform similarly. Figure 4 shows the results. On the Linux cluster, the tests were run on two nodes, with all communication happening across nodes. We ran two cases: one where there were as many processes/threads as the number of processors on a node (four) and one where there were eight processes/threads running on four processors. Both cases show no measurable difference in bandwidth between threads and processes with MPICH2. With Open MPI, there is a decline in bandwidth with threads in the oversubscribed case. On the Sun and IBM SMPs, on the other hand, there is a substantial decline (more than 50% in some cases) in the bandwidth when threads were used instead of processes.

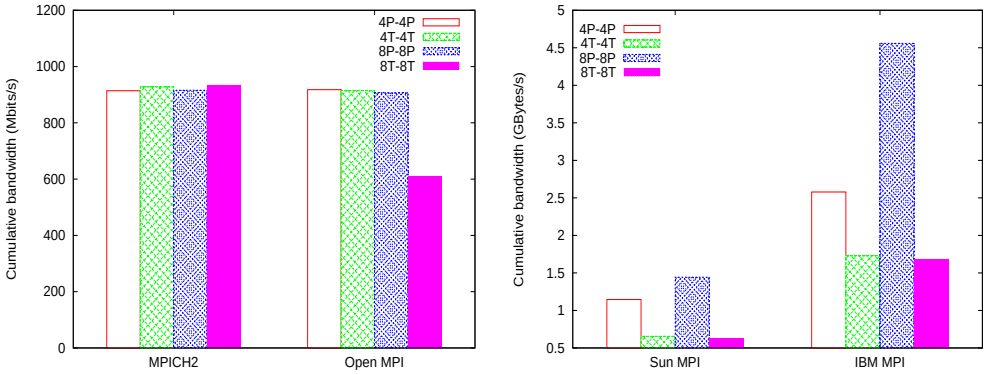

Fig. 4. Concurrent bandwidth test on Linux cluster (left) and Sun and IBM SMPs (right).

We also ran a version of the test that measures the time (latency) for individual short messages instead of concurrent bandwidth for large messages. Figure 5 shows the results. On the Linux cluster with MPICH2, there is a 20 μs overhead in latency when using concurrent threads instead of processes. With Open MPI, the overhead is about 30 μs. With Sun and IBM MPI, the latency with threads is about 10 times the latency with processes.

The overhead of threads is much more noticeable on the shared-memory machines because the overall message-passing performance on those machines is high (very low latency and very high bandwidth). Minimizing the overhead of thread-related locking in such environments is a difficult problem, and more research is needed in this area.

Additional results for several other tests can be found in [23].

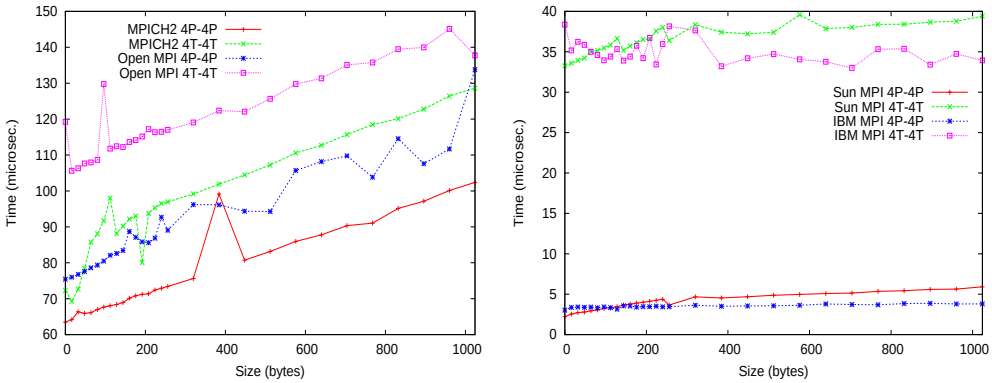

Fig. 5. Concurrent latency test on Linux cluster (left) and Sun and IBM SMPs (right).

5 Summary

Although the MPI standard has existed for a long time and MPI implementations have matured over the years, many areas remain in which MPI implementations still need improvement. Some of these improvements are necessitated by new developments in parallel systems, such as very large scale (more than 100,000 processors) and the advent of multicore chips. Others are just hard topics that need more work. Special efforts are needed in the areas of scalability, fault tolerance, one-sided communication, support for multithreading, and topology awareness. Continued research in these areas and incorporation of research results into production implementations will enable users to take full advantage of the enormous power of the leading supercomputers, which is rapidly approaching 1 petaflop/s.

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 DE-AC02-06CH11357.

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. 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 *Proceedings of SC 2002*. IEEE, 2002.
3. Jehoshua Bruck, Ching-Tien Ho, Schlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, November 1997.
4. Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI Users' Group Meeting*, pages 86–95. Lecture Notes in Computer Science 4192, Springer, September 2006.
5. Surendra Byna, William Gropp, Xian-He Sun, and Rajeev Thakur. Improving the performance of MPI derived datatypes by optimizing memory-access cost. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2003)*, pages 412–419, December 2003.
6. James Cownie and William Gropp. A standard interface for debugger access to message queue information in MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science 1697, Springer-Verlag, pages 51–58, 1999.

7. Graham E. Fagg and Jack J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Recent Advances in Parallel Virutal Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting*, pages 346–353. Lecture Notes in Computer Science 1908, Springer, September 2000.
8. Chris Falzone, Anthony Chan, Ewing Lusk, and William Gropp. Collective error detection for MPI collective operations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, pages 138–147. Lecture Notes in Computer Science 3666, Springer, September 2005.
9. Christopher Gottbrath, Brian Barrett, William D. Gropp, Ewing Lusk, and Jeff Squyres. An interface to support the identification of dynamic MPI-2 processes for scalable parallel debugging. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI Users' Group Meeting*, pages 115–122. Lecture Notes in Computer Science 4192, Springer, September 2006.
10. William Gropp and Rajeev Thakur. Issues in developing a thread-safe MPI implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI Users' Group Meeting*, pages 12–21. Lecture Notes in Computer Science 4192, Springer, September 2006.
11. William Gropp and Rajeev Thakur. Revealing the performance of MPI RMA implementations. Technical Report ANL/MCS-P1419-0507, Mathematics and Computer Science Division, Argonne National Laboratory, May 2007.
12. Intel Trace Analyzer and Collector 7.0 for Linux. `http://www.intel.com`.
13. Hideyuki Jitsumoto, Toshio Endo, and Satoshi Matsuoka. ABARIS: An adaptable fault detection/recovery component framework for MPIs. In *Proceedings of 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS '07) in conjunction with IPDPS 2007*, March 2007.
14. 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.
15. 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.
16. Bettina Krammer, Matthias S. Müller, and Michael M. Resch. MPI application development using the analysis tool MARMOT. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 4th International Conference on Computational Science (ICCS 2004)*, pages 464–471. Lecture Notes in Computer Science 3038, Springer, June 2004.
17. Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.
18. Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Rajeev Thakur, and William Gropp. Formal verification of programs that use MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 13th European PVM/MPI Users' Group Meeting*, pages 30–39. Lecture Notes in Computer Science 4192, Springer, 2006.
19. Hubert Ritzdorf and Jesper Larsson Träff. Collective operations in NEC's high-performance MPI libraries. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, April 2006.
20. ROMIO: A high-performance, portable MPI-IO implementation. `http://www.mcs.anl.gov/romio`.

21. Robert Ross, Neill Miller, and William D. Gropp. Implementing fast and reusable datatype processing. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 404–413. Lecture Notes in Computer Science 2840, Springer, September 2003.
22. 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.
23. Rajeev Thakur and William Gropp. Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. Technical Report ANL/MCS-P1418-0507, Mathematics and Computer Science Division, Argonne National Laboratory, May 2007.
24. Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.
25. Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI-IO. *Parallel Computing*, 28(1):83–105, January 2002.
26. Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *International Journal of High-Performance Computing Applications*, 19(1):49–66, Spring 2005.
27. Rajeev Thakur, Robert Ross, and Robert Latham. Implementing byte-range locks using MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting*, pages 120–129. Lecture Notes in Computer Science 3666, Springer, September 2005.
28. Jesper Larsson Träff. A simple work-optimal broadcast algorithm for message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting*, pages 173–180. Lecture Notes in Computer Science 3241, Springer, September 2004.
29. Jesper Larsson Träff, William Gropp, and Rajeev Thakur. Self-consistent MPI performance requirements. Technical report. Submitted to Euro PVM/MPI 2007, May 2007.
30. Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdoff, and Falk Zimmermann. Flattening on the fly: Efficient handling of MPI derived datatypes. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting*, pages 109–116. Lecture Notes in Computer Science 1697, Springer-Verlag, 1999.
31. Jesper Larsson Träff and Joachim Worringen. Verifying collective MPI calls. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting*, pages 18–27. Lecture Notes in Computer Science 3241, Springer, September 2004.
32. Jeffrey S. Vetter and Bronis R. de Supinski. Dynamic software testing of MPI applications with Umpire. In *Proceedings of SC2000: High Performance Networking and Computing*, pages 70–79, November 2000.