Test Suite for Evaluating Performance of Multithreaded MPI Communication

Rajeev Thakur [a,*] and William Gropp [b]

[a] *Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439, USA*

[b] *Department of Computer Science, University of Illinois, Urbana, IL 61801, USA*

Abstract

As parallel systems are commonly being built out of increasingly large multicore chips, application programmers are exploring the use of hybrid programming models combining MPI across nodes and multithreading within a node. Many MPI implementations, however, are just starting to support multithreaded MPI communication, often focussing on correctness first and performance later. As a result, both users and implementers need some measure for evaluating the multithreaded performance of an MPI implementation. In this paper, we propose a number of performance tests that are motivated by typical application scenarios. These tests cover the overhead of providing the `MPI_THREAD_MULTIPLE` level of thread safety for user programs, the amount of concurrency in different threads making MPI calls, the ability to overlap communication with computation, and other features. We present performance results with this test suite on several platforms (Linux cluster, Sun and IBM SMPs) and MPI implementations (MPICH2, Open MPI, IBM, and Sun).

Key words: Message Passing Interface (MPI), multithreading, performance measurement, benchmarks

1 Introduction

Processor development has headed to an era where chips comprising multiple processors per core are common. As a result, parallel machines are being built

* Corresponding Author
Email addresses: `thakur@mcs.anl.gov` (Rajeev Thakur), `wgropp@uiuc.edu` (William Gropp).

out of multicore chips as the basic building block. With multiple CPUs sharing memory within a single node, application programmers are exploring the use of hybrid programming models comprising MPI across nodes and threads within a node. The threaded portion of the program is implemented either explicitly with a threads library such as Pthreads [5] or implicitly by using compiler directives such as OpenMP [9]. In either case, MPI functions could be called from multiple threads of a process, and efficient support for multithreaded MPI is needed.

MPI implementations, however, have traditionally not provided highly tuned support for multithreaded MPI communication. In fact, many implementations do not even support thread safety (at the time of this writing, Microsoft MPI, SiCortex MPI, NEC MPI, IBM MPI for Blue Gene/L, and Cray MPI for XT4). Other implementations do support thread safety. However, 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 [4]. 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 optimal. Users, therefore, need a way to determine how efficiently an implementation can support multiple threads. Similarly, as implementers experiment with potential performance optimizations, they need a way to measure the outcome. To meet these needs, we have created a test suite that can shed light on the performance of an MPI implementation in the multithreaded case. We describe the tests in the suite, the rationale behind them, and their performance with several MPI implementations (MPICH2, Open MPI, IBM MPI, and Sun MPI) on several platforms.

Related Work. The MPI benchmarks from Ohio State University [10] contain a multithreaded latency test, which is a ping-pong test with one thread on the sender side and two (or more) threads on the receiver side. A number of other MPI benchmarks exist, such as SKaMPI [13] and the Intel MPI Benchmarks [6], but they do not measure the performance of multithreaded MPI programs.

Some research has been done in the area of implementing thread safety in MPI. In [4], we described and analyzed what the MPI Standard says about thread safety and what it implies for an implementation. We also presented an efficient multithreaded algorithm for generating new context ids, which is required for creating new communicators. Protopopov and Skjellum discuss a number of issues related to threads and MPI, including a design for a thread-safe version of MPICH-1 [12,14]. Plachetka describes a mechanism for making a thread-unsafe PVM or MPI implementation quasi-thread-safe by adding an interrupt mechanism and two functions to the implementation [11]. García et

al. present MiMPI, a thread-safe implementation of MPI [3]. TOMPI [2] and TMPI [15] are thread-based MPI implementations, where each MPI rank is actually a thread. (Our paper focuses on conventional MPI implementations where each MPI rank is a process that itself may have multiple threads, all having the same rank.) USFMPI is a multithreaded implementation of MPI that internally uses a separate thread for communication [1]. A good discussion of the difficulty of programming with threads in general is given in [7].

2 Overview of MPI and Threads

To understand the test suite and the rationale behind each test, one must understand the thread-safety specification in MPI [8]. For performance reasons, MPI defines four "levels" of thread safety and allows the user to indicate the level desired—the idea being that the implementation need not incur the cost for a higher level of thread safety than the user needs. The four levels of thread safety are as follows:

(1) `MPI_THREAD_SINGLE` Each process has a single thread of execution.
(2) `MPI_THREAD_FUNNELED` A process may be multithreaded, but only the thread that initialized MPI may make MPI calls.
(3) `MPI_THREAD_SERIALIZED` A process may be multithreaded, but only one thread at a time may make MPI calls.
(4) `MPI_THREAD_MULTIPLE` A process may be multithreaded, and multiple threads may simultaneously call MPI functions (with some restrictions mentioned below).

An implementation is not required to support levels higher than `MPI_THREAD_SINGLE`; that is, an implementation is not required to be thread safe. A fully thread-compliant implementation, however, will support `MPI_THREAD_MULTIPLE`. MPI provides a function, `MPI_Init_thread`, by which the user can indicate the level of thread support desired, and the implementation will return the level supported. A portable program that does not call `MPI_Init_thread` should assume that only `MPI_THREAD_SINGLE` is supported. The tests described in this paper focus on the `MPI_THREAD_MULTIPLE` (fully multithreaded) case.

For `MPI_THREAD_MULTIPLE`, the MPI Standard specifies that when multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order. Also, blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions. MPI also says that it is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls. For example, the user cannot call `MPI_Info_set` and `MPI_Info_free` on

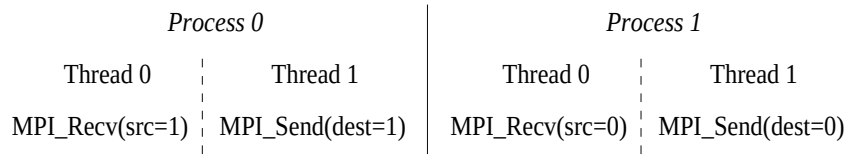

Fig. 1. An implementation must ensure that this example never deadlocks for any ordering of thread execution.

the same `info` object concurrently from two threads of the same process; the user must ensure that the `MPI_Info_free` is called only after `MPI_Info_set` returns on the other thread. Similarly, the user must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads.

A straightforward implication of the MPI thread-safety specification is that an implementation cannot implement thread safety by simply acquiring a lock at the beginning of each MPI function and releasing it at the end of the function: A blocked function that holds a lock may prevent MPI functions on other threads from executing, a situation that in turn might prevent the occurrence of the event that is needed for the blocked function to return. An example is shown in Figure 1. If thread 0 happened to get scheduled first on both processes, and `MPI_Recv` simply acquired a lock and waited for the data to arrive, the `MPI_Send` on thread 1 would not be able to acquire its lock and send its data; hence, the `MPI_Recv` would block forever. Therefore, the implementation must release the lock at least before blocking within the `MPI_Recv` and then reacquire the lock if needed after the data has arrived. (The tests described in this paper provide some information about the fairness and granularity of how blocking MPI functions are handled by the implementation.)

3 The Test Suite

Users of threads in MPI often have the following expectations of the performance of threads, both those making MPI calls and those performing computation concurrently with threads that are making MPI calls.

- The cost of thread safety, compared with lower levels of thread support, such as `MPI_THREAD_FUNNELED`, is relatively low.
- Multiple threads making MPI calls, such as `MPI_Send` or `MPI_Bcast`, can make progress simultaneously.
- A blocking MPI routine in one thread does not consume excessive CPU resources while waiting.

Our tests are designed to test these expectations; in terms of the above categories, they are as follows:

Cost of thread safety One simple test to measure the overhead of `MPI_THREAD_MULTIPLE`.

Concurrent progress Tests to measure concurrent bandwidth by multiple threads of a process to multiple threads of another process, as compared with multiple processes to multiple processes. Both point-to-point and collective operations are included.

Computation overlap Tests to measure the overlap of communication with computation and the ability of the application to use a thread to provide a nonblocking version of a communication operation for which there is no corresponding MPI call, such as nonblocking collectives or I/O operations that involve several steps.

We describe the tests below and present performance results on the following platforms and MPI implementations:

Linux Cluster We used the Breadboard cluster at Argonne, in which each node has two dual-core 2.8 GHz AMD Opteron CPUs. The nodes are connected by Gigabit Ethernet. We used MPICH2 1.0.7 and Open MPI 1.2.6, which are the latest versions of those implementations at the time of this writing. MPICH2 was configured with the default options, which result in thread safety being enabled only if the user initializes MPI by calling `MPI_Init_thread` with `MPI_THREAD_MULTIPLE`. For Open MPI, the process-only tests used the default build, which does not support multithreading. The threaded tests used a build configured with `--enable-mpi-threads`. A third build with the additional option `--enable-progress-threads` was used only for some tests in Section 3.6 (described further in that section). Open MPI has a disclaimer that support for thread safety is not well tested in the v1.2 series; nonetheless, we did not experience any crashes when running with Open MPI.

Sun T5120 Server We used a Sun T5120 server from the Sun cluster at the RWTH Aachen University. The specific machine we ran on was a Sun T5120 with eight 1.4 GHz UltraSPARC T2 ("Niagara2") cores. It runs Sun's MPI (ClusterTools 6).

IBM SMP We also used an IBM p655+ SMP from the DataStar cluster at the San Diego Supercomputer Center. The machine has eight 1.7 GHz POWER4+ CPUs and runs IBM's MPI.

We note that the results presented in the following sections are heavily implementation dependent and may change substantially with newer versions of the implementations. The goal of this paper is not to compare the different implementations, but to illustrate the results provided by the test suite and how these results should be analyzed.

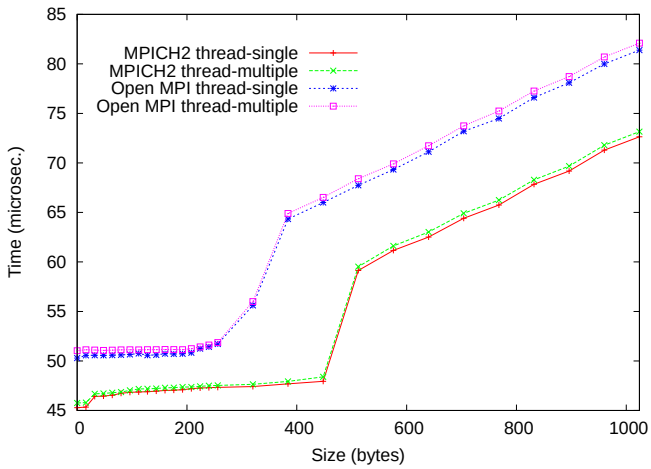

Fig. 2. Overhead of `MPI_THREAD_MULTIPLE` on the Linux cluster.

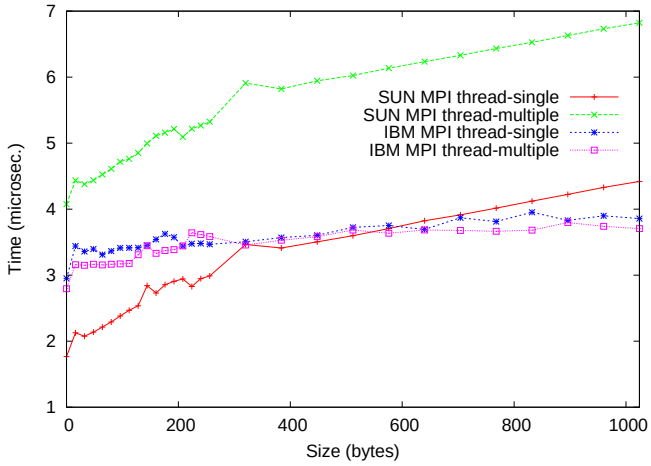

Fig. 3. Overhead of `MPI_THREAD_MULTIPLE` on Sun and IBM SMPs.

3.1 Overhead with `MPI_THREAD_MULTIPLE`

Our first test measures the ping-pong latency for two cases of a *single*-threaded program: initializing MPI with just `MPI_Init` and initializing it with `MPI_Init_thread` for `MPI_THREAD_MULTIPLE`. In the latter case, the implementation must assume the program is multithreaded and may call MPI functions from any thread (even though this test does not). The test demonstrates the overhead of ensuring thread safety for the `MPI_THREAD_MULTIPLE` case, which is typically implemented by acquiring and releasing mutex locks.

Figures 2 and 3 show the results. On the Linux cluster, with both MPICH2 and Open MPI, the overhead of `MPI_THREAD_MULTIPLE` is less than 0.5 μs. On the IBM SMP with IBM MPI, it is negligible. On the other hand, on the Sun SMP with Sun MPI, the overhead is much higher—around 2.5 μs.

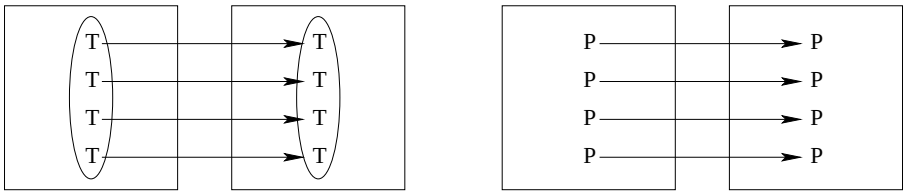

Fig. 4. Communication test when using multiple threads (left) versus multiple processes (right).

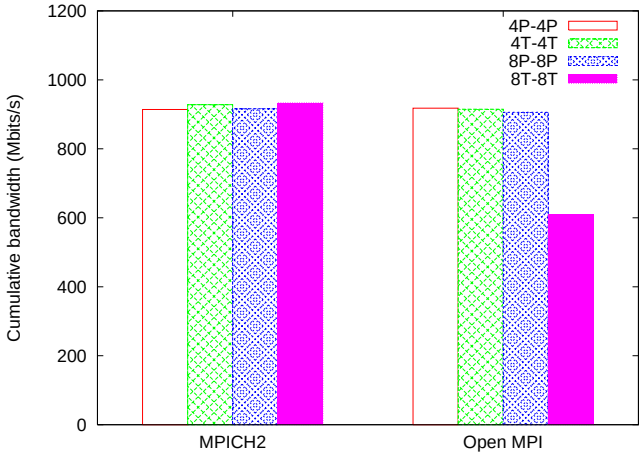

Fig. 5. Concurrent bandwidth test on Linux cluster.

3.2 Concurrent Bandwidth

The second test measures the cumulative bandwidth obtained when multiple threads of a process communicate with multiple threads of another process compared with multiple processes instead of threads (see Figure 4). It demonstrates how much thread locks affect the cumulative bandwidth; ideally, the multiprocess and multithreaded cases should perform similarly.

Figure 5 and 6 show 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. In both cases, there is 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. It is harder to provide low overhead in these shared-memory environments because the communication bandwidths are so high.

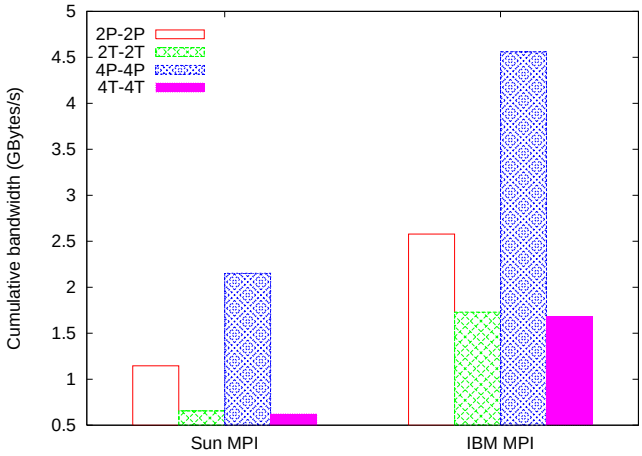

Fig. 6. Concurrent bandwidth test on Sun and IBM SMPs.

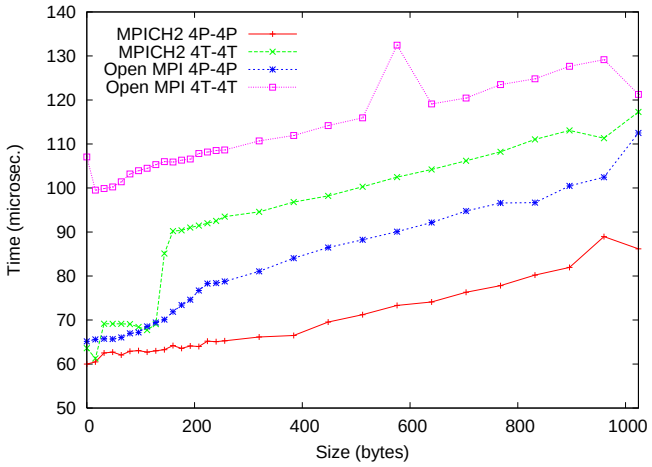

Fig. 7. Concurrent latency test on Linux cluster.

3.3 Concurrent Latency

Our third test is similar to the concurrent bandwidth test except that it measures the time for individual short messages instead of concurrent bandwidth for large messages. Figures 7 and 8 show the results. On the Linux cluster with both MPICH2 and Open MPI, there is about 30 μs overhead in latency when using concurrent threads instead of processes. On the Sun and IBM SMPs, the latency with concurrent processes is very low—in the range of 2–4 μs. Comparatively, the overhead with threads is very high: On the Sun SMP, the latency with threads rises to 10–16 μs; on the IBM system it shoots to 33–38 μs. Providing low overhead with threads is more challenging on these systems because the basic message-passing latency itself is very low. Careful design and tuning of code is needed to minimize the overhead.

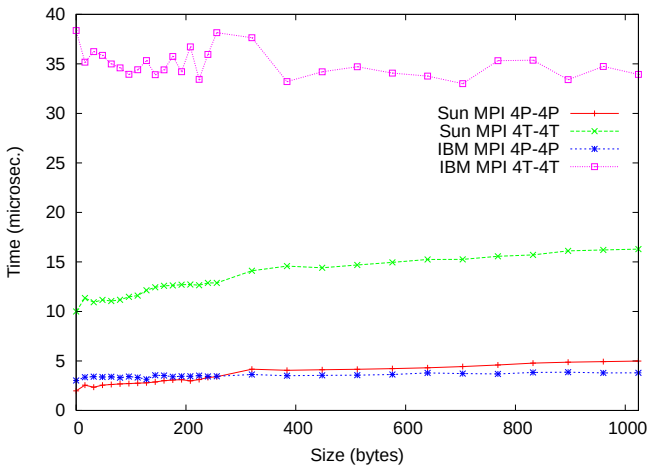

Fig. 8. Concurrent latency test on Sun and IBM SMPs.

3.4 Message Rate

This test is similar to the concurrent latency test except that it measures the message rate for zero-byte sends. The message rate for each thread/process is calculated as the reciprocal of the average latency for zero-byte sends observed by that thread/process. The sender waits for a reply only once every 256 iterations, so it is not a ping-pong test. The individual message rates are summed to determine the total message rate. As before, the case where multiple threads of a process communicate with multiple threads of another process is compared with processes communicating with processes.

Table 1 shows the message rate on the Linux cluster. The difference between the 1T–1T and 1P–1P cases is that the former uses `MPI_Init_thread` with `MPI_THREAD_MULTIPLE` and incurs the associated costs, whereas the latter just calls `MPI_Init`. With both MPICH2 and Open MPI, the message rate is only slightly lower when there is one thread versus one process, but it declines substantially in the four-threads case, indicating locking overhead.

Table 2 shows the message rate on the Sun and IBM SMPs. Here, the overall message rates are much higher because all communication takes place within a node using shared memory. On the Sun machine, the message rate is substantially lower with threads, both in the one thread and four threads cases. On the IBM system, the performance is good for one thread, but drops very sharply in the four-threads case.

Table 1
Message rate on Linux cluster (messages/sec)

	1P–1P	1T–1T	4P–4P	4T–4T
MPICH2	322,892	290,790	637,486	184,191
Open MPI	285,672	212,657	613,333	144,131

Table 2
Message rate on Sun and IBM SMPs (messages/sec)

	1P-1P	1T-1T	4P-4P	4T-4T
Sun MPI	762,251	262,239	2,845,098	465,120
IBM MPI	342,957	342,845	1,355,445	125,988

3.5 Concurrent Short-Long Messages

The fourth test is a blend of the concurrent bandwidth and concurrent latency tests. It has two versions. In the threads version, rank 0 has two threads: one sends a long message to rank 1, and the other sends a series of short messages to rank 2. The second version of the test is similar except that the two senders are processes instead of threads. This test tests the fairness of thread scheduling and locking. If they were fair, one would expect each of the short messages to take roughly the same amount of time.

The results are shown in Figures 9 and 10. ("Iteration" on the X-axis refers to the iteration count of the loop that sends a series of short messages, one per iteration.) With both MPICH2 and Open MPI, the cost of communicating the long message is evenly distributed among a number of short messages. A single short message is not penalized for the entire time the long message is communicated. This result demonstrates that, in the threaded case, locks are fairly held and released and that the thread blocked in the long-message send does not block the other thread. With Sun and IBM MPI, however, one sees spikes in the graphs. This behavior may be because these implementations use memory copying to communicate data, and it is harder to overlap this memory-copy time with the memory copying on the other thread.

3.6 Computation/Communication Overlap

Our fifth test measures the ability of an implementation to overlap communication with computation and provides users an alternative way of achieving such an overlap if the implementation does not do so. The test has two versions. The first version has an iterative loop in which a process communicates with its four nearest neighbors by posting nonblocking sends and receives,

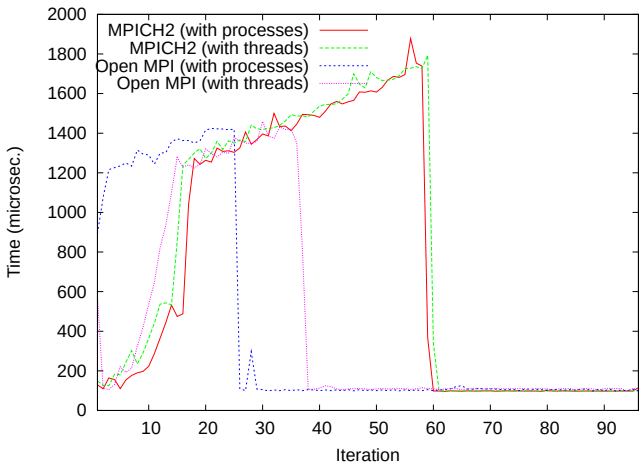

Fig. 9. Concurrent short-long messages test on Linux cluster.

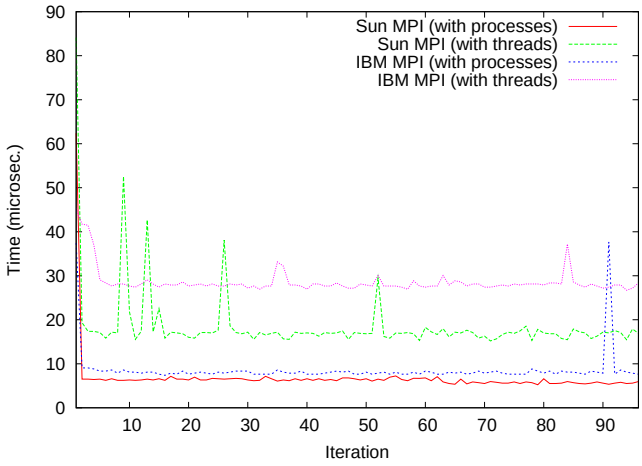

Fig. 10. Concurrent short-long messages test on Sun and IBM SMPs.

followed by a computation phase, followed by an `MPI_Waitall` for the communication to complete. The second version is similar except that, before the iterative loop, each process spawns a thread that blocks on an `MPI_Recv`. The matching `MPI_Send` is called by the main thread only at the end of the program, just before `MPI_Finalize`. The thread thus blocks in the `MPI_Recv` while the main thread is in the communication-computation loop. Since the thread is executing an MPI function, whenever it gets scheduled by the operating system, it can cause progress to occur on the communication initiated by the main thread. This technique effectively simulates asynchronous progress by the MPI implementation. If the total time taken by the communication-computation loop in this case is less than that in the nonthreaded version, it indicates that the MPI implementation on its own does not overlap communication with computation.

Figures 11 and 12 shows the results. Here, "no overlap" refers to the test without the thread, and "overlap" refers to the test with the thread. The

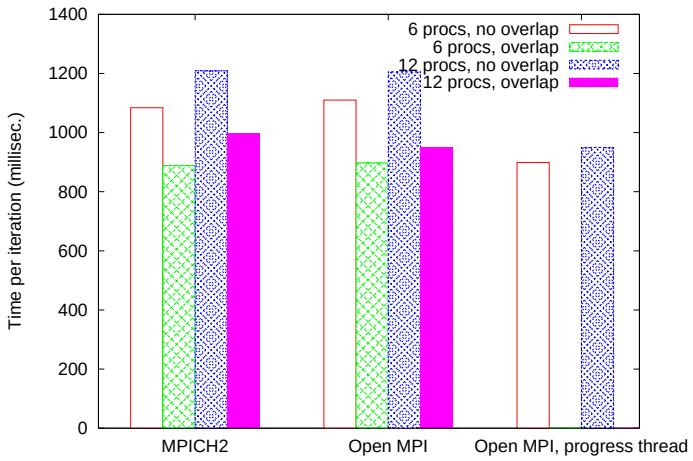

Fig. 11. Computation/communication overlap test on Linux cluster.

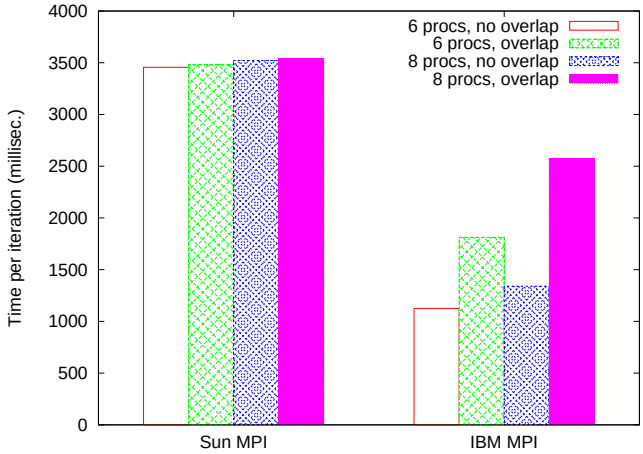

Fig. 12. Computation/communication overlap test on Sun and IBM SMPs.

results with MPICH2 demonstrate no asynchronous progress, as the overlap version of the test performs better. With Open MPI, we ran two experiments. We first used the default build; the results indicate that it performs similarly to MPICH2—no overlap of computation and communication. Open MPI can also be optionally built to use an extra thread internally for asynchronous progress. With this version of the library, we see that indeed there is asynchronous progress, as the performance is nearly the same as for the "overlap" test with the default build. That is, the case with the implementation-created progress thread performs similarly to the case with the user-created thread.

We note that always using an extra thread for progress has other performance implications. For example, it can result in higher communication latencies because of the thread-switching overhead. Due to lack of space, we did not run all the other tests with the version of Open MPI configured to use an extra thread.

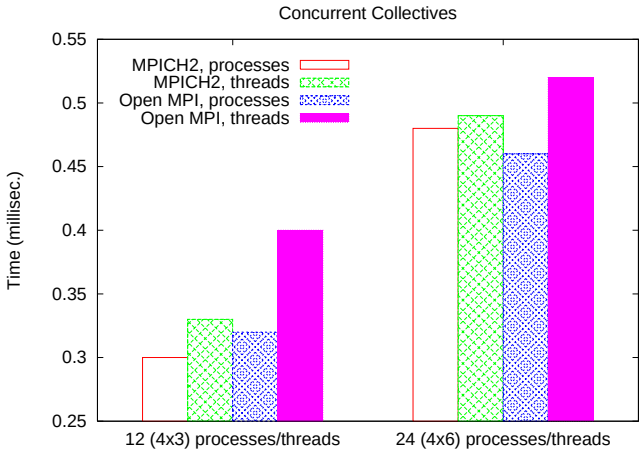

Fig. 13. Concurrent collectives test on the Linux cluster (4x3 refers to 4 process/threads each on 3 nodes).

The results on the Sun and IBM SMPs indicate no overlap. In fact, on the IBM machine, the performance was worse with the overlap thread because of the higher overhead when using threads.

3.7 Concurrent Collectives

Our sixth test compares the performance of concurrent calls to a collective function (`MPI_Allreduce`) issued from multiple threads to that when issued from multiple processes. The test uses multiple communicators, and processes are arranged such that the processes belonging to a given communicator are located on different nodes. In other words, collective operations are issued by multiple threads/processes on a node, with all communication taking place across nodes (similar to Figure 4 but for collectives and using multiple nodes).

Figure 13 shows the results on the Linux cluster. MPICH2 has relatively small overhead for the threaded version, compared with Open MPI.

3.8 Concurrent Collectives and Computation

Our final test evaluates the ability to use a thread to hide the latency of a collective operation while using all available processors to perform computations. It uses p+1 threads on a node with p processors. Threads 0–$(p-1)$ perform some computation iteratively. Thread p does an `MPI_Allreduce` with its corresponding threads on other nodes. When the allreduce completes, it sets a flag, which stops the iterative loop on the other threads. The average number of iterations completed on the threads is reported. This number is

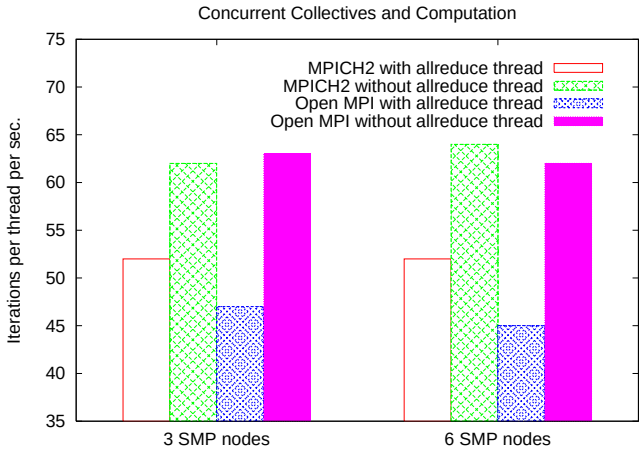

Fig. 14. Concurrent collectives and computation test on the Linux cluster.

compared with the case with no allreduce thread (the higher the better).

Figure 14 shows the results on the Linux cluster. MPICH2 demonstrates a better ability than Open MPI to hide the latency of the allreduce.

4 Conclusions

Supporting thread safety in MPI is not trivial or straightforward, and achieving good performance requires careful design and implementation. As increasing numbers of MPI implementations support fully multithreaded MPI communication, it is essential to have some measure of evaluating how efficient they are. We have developed a test suite that provides such a quantitative measure. We presented its performance on multiple platforms and implementations. The results indicate relatively good performance with MPICH2 and Open MPI on the Linux cluster. The relatively slower communication method (TCP over Gigabit Ethernet) masks some of the overheads associated with maintaining thread safety. The performance results on the Sun and IBM SMPs demonstrate that threading overhead is much more noticeable on such systems because the overall communication through shared memory is very fast. Significant research and development is needed to minimize threading overhead in MPI on such platforms.

This test suite will continue to be extended and new tests will be added, such as to measure the overlap of computation/communication with the MPI-2 file I/O and connect-accept features. We will also accept contributions from others to the test suite. The test suite can be downloaded from `www.mcs.anl.gov/~thakur/thread-tests`.

Acknowledgments

We thank the RWTH Aachen University and the San Diego Supercomputer Center for providing computing time on their systems. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] Sadik G. Caglar, Gregory D. Benson, Qing Huang, and Cho-Wai Chu. USFMPI: A multi-threaded implementation of MPI for Linux clusters. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and Systems*, 2003.

[2] Erik D. Demaine. A threads-only MPI implementation for the development of parallel programs. In *Proceedings of the 11th International Symposium on High Performance Computing Systems*, pages 153–163, July 1997.

[3] Francisco García, Alejandro Calderón, and Jesús Carretero. MiMPI: A multithread-safe implementation of MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting*, pages 207–214. Lecture Notes in Computer Science 1697, Springer, September 1999.

[4] William Gropp and Rajeev Thakur. Thread safety in an MPI implementation: Requirements and analysis. *Parallel Computing*, 33(9):595–604, September 2007.

[5] IEEE/ANSI Std. 1003.1. Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API) [C Language], 1996 edition.

[6] Intel MPI benchmarks. `http://www.intel.com`.

[7] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[8] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.

[9] OpenMP. `http://www.openmp.org`.

[10] OSU MPI benchmarks. `http://mvapich.cse.ohio-state.edu/benchmarks`.

[11] Tomas Plachetka. (Quasi-) thread-safe PVM and (quasi-) thread-safe MPI without active polling. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting*, pages 296–305. Lecture Notes in Computer Science 2474, Springer, September 2002.

[12] Boris V. Protopopov and Anthony Skjellum. A multithreaded message passing interface (MPI) architecture: Performance and program issues. *Journal of Parallel and Distributed Computing*, 61(4):449–466, April 2001.

[13] Ralf Reussner, Peter Sanders, and Jesper Larsson Träff. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65, January 2002.

[14] Anthony Skjellum, Boris Protopopov, and Shane Hebert. A thread taxonomy for MPI. In *Proceedings of the 2nd MPI Developers Conference*, pages 50–57, June 1996.

[15] Hong Tang and Tao Yang. Optimizing threaded MPI execution on SMP clusters. In *Proceedings of the 15th ACM International Conference on Supercomputing*, pages 381–392, June 2001.