Test Suite for Evaluating Performance of MPI Implementations That Support MPI_THREAD_MULTIPLE

Rajeev Thakur and William Gropp

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

Abstract. MPI implementations that support the highest level of thread safety for user programs, `MPI_THREAD_MULTIPLE`, are becoming widely available. Users often expect that different threads can execute independently and that the MPI implementation can provide the necessary level of thread safety with only a small overhead. The MPI Standard, however, requires only that no MPI call in one thread block MPI calls in other threads; it makes no performance guarantees. Therefore, some way of measuring an implementation's performance is needed. 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).

1 Introduction

With thread-safe MPI implementations becoming increasingly common, users are able to write multithreaded MPI programs that make MPI calls concurrently from multiple threads. 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 [2]. 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. Users, therefore, need a way to determine how efficiently an implementation can support multiple threads. Similarly, as implementers experiment with a potential performance optimization, they need a way to measure the outcome. (We ourselves face this situation in MPICH2.) To meet these needs, we have created a test suite that can shed light

F. Cappello et al. (Eds.): EuroPVM/MPI 2007, LNCS 4757, pp. 46–55, 2007.
© Springer-Verlag Berlin Heidelberg 2007

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 [4] 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 [6] and the Intel MPI Benchmarks [3], but they do not measure the performance of multithreaded MPI programs. A good discussion of the issues in developing a thread-safe MPI implementation is given in [2]. Other thread-safe MPI implementations are described in [1,5].

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. 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,

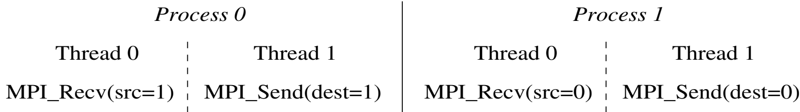

Fig. 1. An implementation must ensure that this example never deadlocks for any ordering of thread execution

the user cannot call `MPI_Info_set` and `MPI_Info_free` on 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 `MPI_THREAD_MULTIPLE` overhead.

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 1.0.5 and Open MPI 1.2.1.

Sun Fire SMP. We used a Sun Fire SMP from the Sun cluster at the RWTH Aachen University. The specific machine we ran on was a Sun Fire E2900 with eight dual-core UltraSPARC IV 1.2 GHz CPUs. It runs Sun's MPI (ClusterTools 5).

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.

3.1 MPI_THREAD_MULTIPLE Overhead

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`. This test demonstrates the overhead of ensuring thread safety for `MPI_THREAD_MULTIPLE`—typically implemented by acquiring and releasing locks—even though no special steps are needed in this case because the process is single threaded (but the implementation does not know that).

Figure 2 shows 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 less than 0.25 μs. On the other hand, on the Sun SMP with Sun MPI, the overhead is very high—more than 3 μs.

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 3). It 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

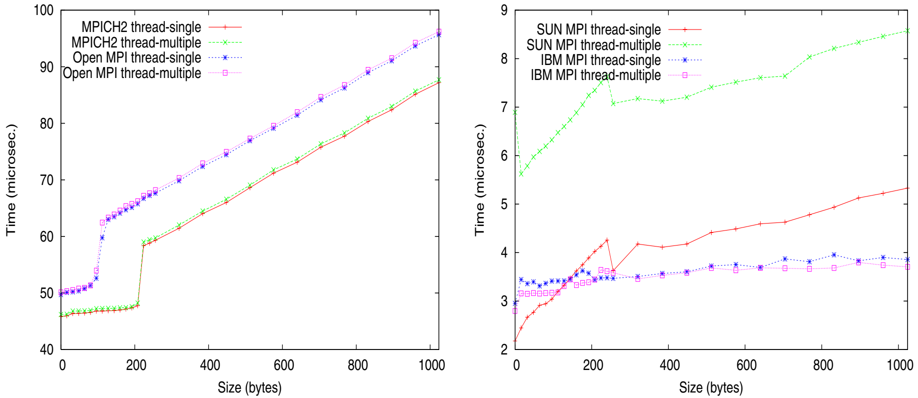

Fig. 2. Overhead of `MPI_THREAD_MULTIPLE` on the Linux cluster (left) and Sun and IBM SMPs (right)

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. Although it is harder to provide low overhead in these shared-memory environments because the communication bandwidths are so high, we believe a tuned implementation should be able to support concurrent threads with a much smaller loss in performance.

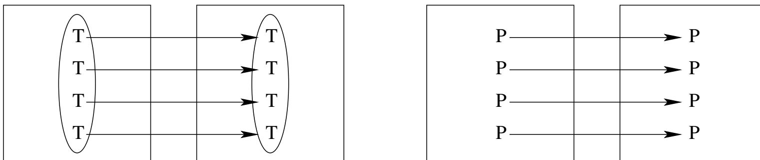

Fig. 3. Communication test when using multiple threads (left) versus multiple processes (right)

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. 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.

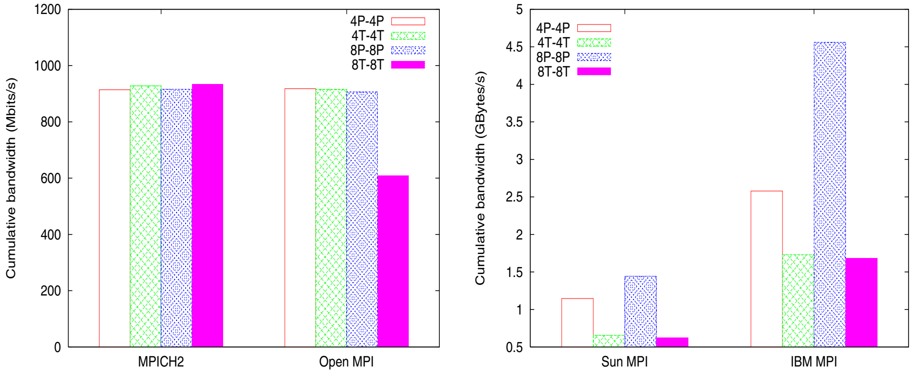

Fig. 4. Concurrent bandwidth test on Linux cluster (left) and Sun and IBM SMPs (right)

Again, although it is more difficult to provide low overhead on these machines because the basic message-passing latency is so low, a tuned implementation should be able to do better than a factor of 10 higher latency in the threaded case.

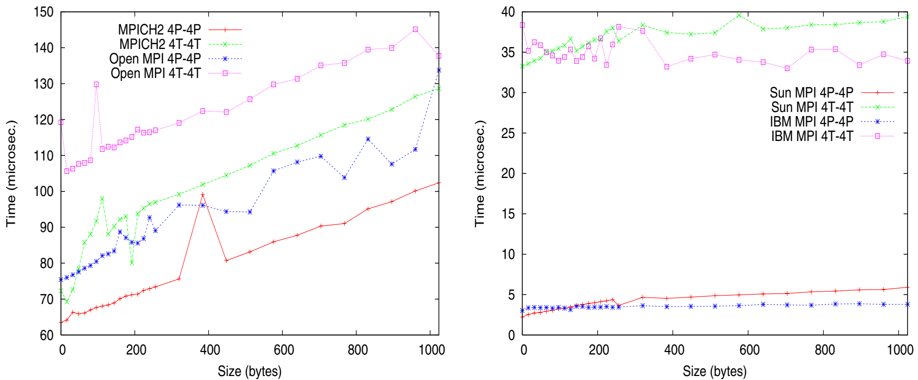

Fig. 5. Concurrent latency test on Linux cluster (left) and Sun and IBM SMPs (right)

3.4 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 Figure 6. With both MPICH2 and Open MPI, the cost of communicating the long message is evenly distributed among a number

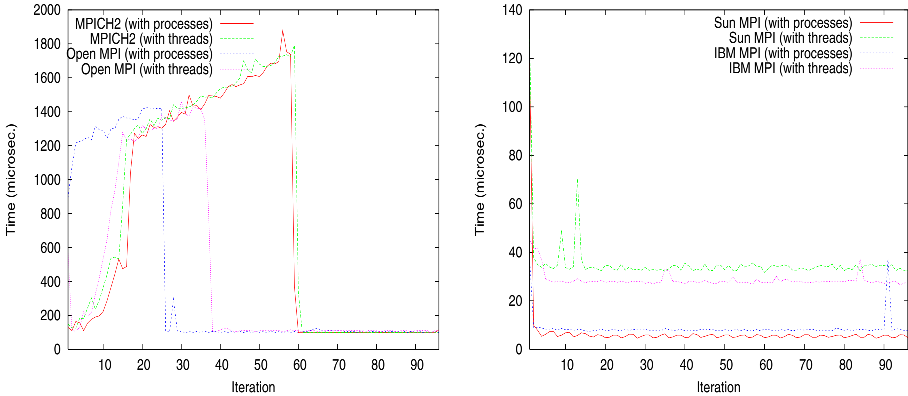

Fig. 6. Concurrent short-long messages test on Linux cluster (left) and Sun and IBM SMPs (right)

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.

3.5 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, 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.

Figure 7 shows the results. Here, "no overlap" refers to the test without the thread, and "overlap" refers to the test with the thread. The results with MPICH2 demonstrate no asynchronous progress, as the overlap version of the

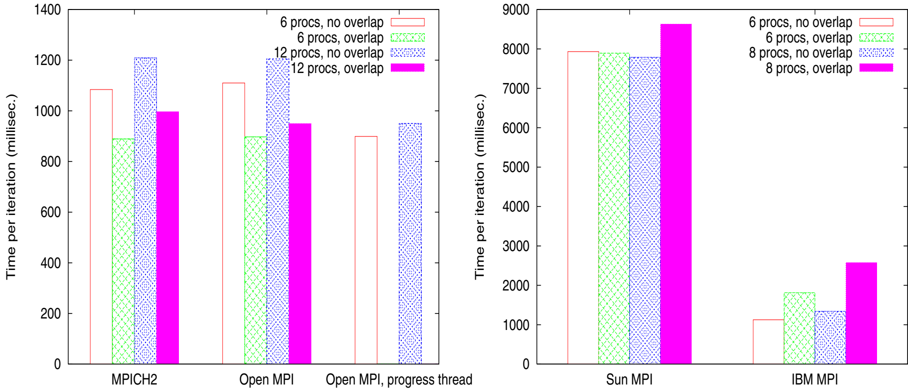

Fig. 7. Computation/communication overlap test on Linux cluster (left) and Sun and IBM SMPs (right)

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.

The results on the Sun and IBM SMPs indicate no overlap. In fact, with eight processes, the performance was worse with the overlap thread because of the high overhead when using threads with these MPI implementations.

3.6 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 3 but for collectives and using multiple nodes).

Figure 8 (left) shows the results on the Linux cluster. MPICH2 has relatively small overhead for the threaded version, compared with Open MPI.

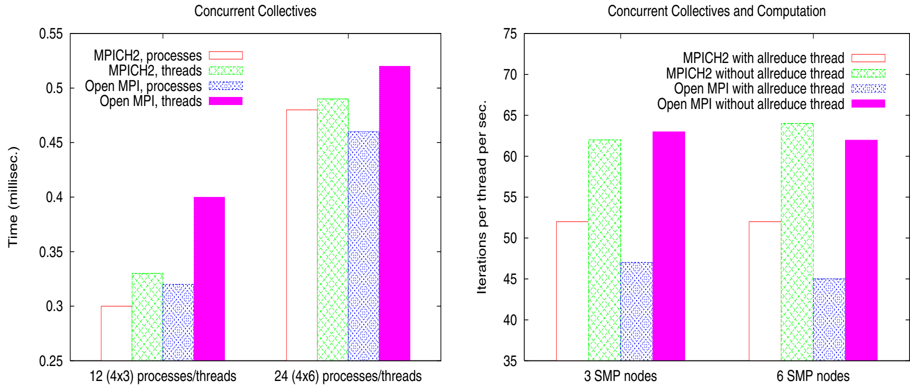

Fig. 8. Left: Concurrent collectives test on the Linux cluster (4x3 refers to 4 process/threads each on 3 nodes). Right: Concurrent collectives and computation test on the Linux cluster.

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

Figure 8 (right) shows the results on the Linux cluster. MPICH2 demonstrates a better ability than Open MPI to hide the latency of the allreduce.

4 Concluding Remarks

As MPI implementations supporting `MPI_THREAD_MULTIPLE` become increasingly available, there is a need for tests that can shed light on the performance and overhead associated with using multiple threads. We have developed such a test suite and presented its performance on multiple platforms and implementations. The results indicate relatively good performance with MPICH2 and Open MPI on Linux clusters, but poor performance with IBM and Sun MPI on IBM and Sun SMP systems.

We plan to add more tests to the suite, 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 will be available for download 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 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. García, F., Calderón, A., Carretero, J.: MiMPI: A multithread-safe implementation of MPI. In: Margalef, T., Dongarra, J.J., Luque, E. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 1697, pp. 207–214. Springer, Heidelberg (1999)
2. Gropp, W., Thakur, R.: Issues in developing a thread-safe MPI implementation. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface. LNCS, vol. 4192, pp. 12–21. Springer, Heidelberg (2006)
3. Intel MPI benchmarks, `http://www.intel.com`
4. OSU MPI benchmarks, `http://mvapich.cse.ohio-state.edu/benchmarks`
5. Protopopov, B.V., Skjellum, A.: A multithreaded message passing interface (MPI) architecture: Performance and program issues. Journal of Parallel and Distributed Computing 61(4), 449–466 (2001)
6. Reussner, R., Sanders, P., Träff, J.L.: SKaMPI: A comprehensive benchmark for public benchmarking of MPI. Scientific Programming 10(1), 55–65 (2002)