Toward Efficient Support for Multithreaded MPI Communication

Pavan Balaji[1], Darius Buntinas[1], David Goodell[1],
William Gropp[2], and Rajeev Thakur[1]

[1] Mathematics and Computer Science Division,
Argonne National Laboratory, Argonne, IL 60439, USA
[2] Department of Computer Science,
University of Illinois, Urbana, IL, 61801, USA

Abstract. To make the most effective use of parallel machines that are being built out of increasingly large multicore chips, researchers are exploring the use of programming models comprising a mixture of MPI and threads. Such hybrid models require efficient support from an MPI implementation for MPI messages sent from multiple threads simultaneously. In this paper, we explore the issues involved in designing such an implementation. We present four approaches to building a fully thread-safe MPI implementation, with decreasing levels of critical-section granularity (from coarse-grain locks to fine-grain locks to lock-free operations) and correspondingly increasing levels of complexity. We describe how we have structured our implementation to support all four approaches and enable one to be selected at build time. We present performance results with a message-rate benchmark to demonstrate the performance implications of the different approaches.

1 Introduction

Processor development is clearly heading to an era where chips comprising multiple processor cores (tens or even hundreds) are ubiquitous. As a result, parallel systems are increasingly being built with multiple CPU cores on a single node, all sharing memory, and the nodes themselves are connected by some kind of interconnection network. On such systems, it is of course possible to run applications as pure MPI processes, one per core. However, as the total number of processes gets very large, the local problem size per process in some applications may decrease to a level where the program does not scale any further. Also, on some systems, running multiple MPI processes per node may restrict the amount of resources, such as TLB space or memory, available to each process. To alleviate these problems, researchers are evaluating other programming models that involve fewer MPI processes per node and use threads to exploit loop-level and other parallelism. Such a hybrid model can be achieved by either explicitly writing a multithreaded MPI program, using say POSIX threads (Pthreads), or by augmenting an MPI program with OpenMP directives [15]. In either case, MPI functions could be called from multiple threads of a process.

MPI implementations have traditionally not provided highly tuned support for multithreaded MPI communication. In fact, many implementations do not even support thread safety. For example, the versions of the following MPI implementations available at the time of this writing do not support thread safety: Microsoft MPI, SiCortex MPI, NEC MPI, IBM MPI for Blue Gene/L, Cray MPI for XT4, and Myricom's MPICH2-MX. Other MPI implementations, such as MPICH2, Open MPI, MVAPICH2, IBM MPI for Blue Gene/P and Power systems, and Intel, HP, SGI, and SUN MPIs do support thread safety. With the increasing use of threads, just supporting thread safety is not sufficient—*efficient* support for multithreaded MPI is needed. Designing an efficient, thread-safe MPI implementation is a nontrivial task. Several issues must be considered, as outlined in [6]. In this paper, we describe our efforts at improving the multithreaded support in our MPI implementation, MPICH2 [10]. We present four approaches to building a fully thread-safe MPI implementation, with decreasing levels of critical-section granularity and correspondingly increasing levels of complexity. We describe how we have structured our implementation to support all four approaches and enable one to be selected at build time. We present performance results with a message-rate benchmark to demonstrate the performance implications of the different approaches.

Related Work The issue of efficiently supporting multithreaded MPI communication has received only limited attention in the literature. In [6], 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, 13]. 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 [5]. TOMPI [3] and TMPI [14] 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 [2]. A good discussion of the difficulty of programming with threads in general is given in [8].

2 Thread Safety in MPI

For performance reasons, MPI defines four "levels" of thread safety [9] 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:

Process 0		*Process 1*	
Thread 0	Thread 1	Thread 0	Thread 1
MPI_Recv(src=1)	MPI_Send(dest=1)	MPI_Recv(src=0)	MPI_Send(dest=0)

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

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 a few restrictions mentioned below).

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. This paper focuses 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. (The example in Figure 1 must not deadlock for any ordering of thread execution.) 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 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.

3 Choices of Critical-Section Granularity

To support multithreaded MPI communication, the implementation must protect certain data structures and portions of code from being accessed by multiple threads simultaneously in order to avoid race conditions. A portion of code so protected is called a *critical section* [4]. The granularity (size) of the critical section and the exact mechanism used to implement the critical section can have a significant impact on performance. In general, having smaller critical sections allows more concurrency among threads but incurs the cost of frequently acquiring and releasing the critical section. A critical section can be implemented either by using mutex locks or in a lock-free manner by using assembly-level atomic operations such as compare-and-swap or fetch-and-add [7]. Mutex locks are comparatively expensive, whereas atomic operations are non-portable and can make the code more complex.

We describe four approaches to the selection of critical-section granularity in a thread-safe MPI implementation.

Global There is a single, global[3] critical section that is held for the duration of most MPI functions, except if the function is going to block on a network operation. In that case, the critical section is released before blocking and then reacquired after the network operation returns. A few MPI functions have no thread-safety implications and hence have no critical section (e.g., `MPI_Wtime`) [1, 6]. This is the simplest approach and is used in the past few releases of MPICH2.

Brief Global There is a single, global critical section, but it is held only when required. This approach permits concurrency between threads making MPI calls, except when common internal data structures are being accessed. However, it is more difficult to implement than Global, because determining where a critical section is needed, and where not, requires care.

Per Object There are separate critical sections for different objects and classes of objects. For example, there may be a separate critical section for communication to a particular process. This approach permits even more concurrency between threads making MPI calls, particularly if the underlying communication system supports concurrent communication to different processes. Correspondingly, it requires even more care in implementing.

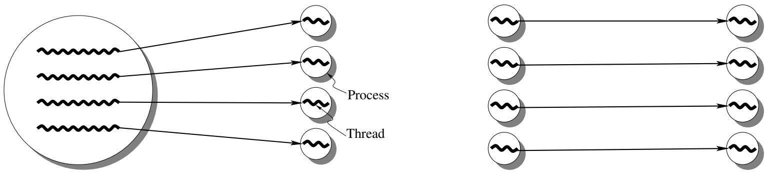
Lock Free Instead of critical sections, lock-free (or wait-free) synchronization methods [7] are implemented by using atomic operations that exploit processor-specific features. This approach offers the potential for improved performance and greater concurrency. Complexity-wise, it is the hardest of the four.

To manage building and experimenting with these four options in MPICH2, we have developed a set of abstractions built around named critical sections and related concepts. These are implemented as compile-time macros, ensuring that there is no extra overhead. Each section of code that requires atomic access to shared data structures is enclosed in a begin/end of a named critical section. In addition, the particular object (if relevant) is passed to the critical section. For example,

```
MPIU_THREAD_CS_BEGIN(COMM,vc)
... code to access a virtual communication channel vc
MPIU_THREAD_CS_END(COMM,vc)
```

In the Global mode, there is an "`ALLFUNC`" (all functions) critical section, and the other macros, such as the `COMM` one above, are defined to be empty so that there is no extra overhead. In the Brief-Global mode, the `ALLFUNC` critical

[3] Global here means global to all threads of a process.



(a) Threads

(b) Processes

Fig. 2. Illustration of test programs. Multiple threads or processes send messages to a different single-threaded receiving process.

section is defined to be empty, and others, such as the above `COMM` critical section, are defined to acquire and release a common, global mutex. The `vc` argument to the macro is ignored in that case. In the Per-Object mode, the situation is similar to that in Brief Global, except that instead of using a common, global mutex, the critical-section macro uses a mutex that is part of the object passed as the second argument of the macro.

For Lock Free, a different code path must be followed. To help with this case, we have implemented a portable library of atomic operations (such as compare-and-swap, test-and-set, fetch-and-add) that are implemented separately for different architectures by using assembly-language instructions. By using these atomic operations, we can replace many of the critical sections with lock-free code. (This part of the code is still under development.)

4 Performance Experiments

To assess the performance of each granularity option, we wrote a test that measures the message rate achieved by n threads of a process sending to n single-threaded receiving processes, as shown in Figure 2(a). The receiving processes prepost 128 receives using `MPI_Irecv`, send an acknowledgment to the sending threads, and then wait for the receives to complete. After receiving the acknowledgment, the threads of the sending process send 128 messages using `MPI_Send`. This process is repeated for 100,000 iterations. The acknowledgment message in each iteration ensures that the receives are posted before the sends arrive, so that there are no unexpected messages. The sending process calls `MPI_Init_thread` with the thread level set to `MPI_THREAD_MULTIPLE` (even for runs with only one thread, in order to show the overhead of providing thread safety). The message rate is calculated as $n/avg\_latency$, where n is the number of sending threads or processes, and $avg\_latency$ is $avg\_looptime/(niters * 128)$, where $avg\_looptime$ is the execution time of the entire iteration loop averaged over the sending threads.

To provide a baseline message rate, we also measured the message rate achieved with separate processes (instead of threads) for sending. For this purpose, we used a modified version of the test that uses multiple single-threaded sending processes, as shown in Figure 2(b). The sending processes simply call `MPI_Init`, which sets the thread level to `MPI_THREAD_SINGLE`.

We performed three sets of experiments to measure the impact of critical-section granularity. The first set does not perform any actual communication (send to MPI_PROC_NULL), the second does blocking communication, and the third does nonblocking communication.

The tests were run on a single Linux machine with two 2.6 GHz, quad-core Intel Clovertown chips (total 8 cores), with our development version of MPICH2 in which the ch3:sock (TCP) channel was modified to incorporate the thread-safety approaches described in this paper.

4.1 Performance with MPI_PROC_NULL

This test is intended to measure the threading overhead in the MPICH2 code in the absence of any network communication. For this purpose, we use MPI_PROC_NULL as the destination in MPI_Send and as a source in MPI_Irecv. In MPICH2, an MPI_Send to MPI_PROC_NULL is handled at a layer above the device-specific code and does not involve manipulation of any shared data structures.

Figure 3 shows the aggregate message rate of the sending threads or processes as a function of the number of threads or processes. In the multiple-process case, the message rate increases with the number of senders because there is no contention for critical sections. In the multithreaded case with Brief Global, the performance is almost identical to multiple processes because Brief Global acquires critical sections only as needed, and in this case no critical section is needed as there is no communication. With the Global mode, however, there is a significant decline in message rate because, in this mode, a critical section is acquired on entry to an MPI function, which serializes the accesses by different threads.

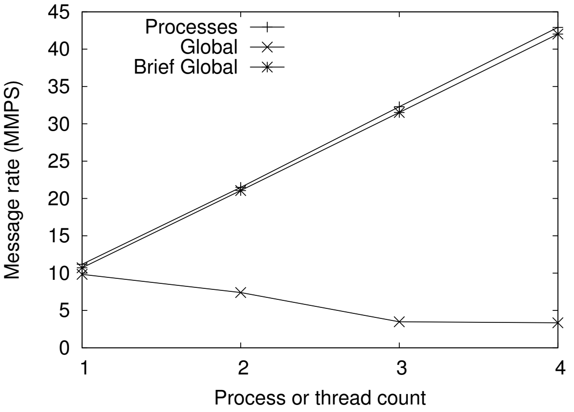


Fig. 3. Message rate (in million messages per sec.) for a multithreaded process sending to MPI_PROC_NULL with Global and Brief-Global granularities, compared to that with multiple processes.

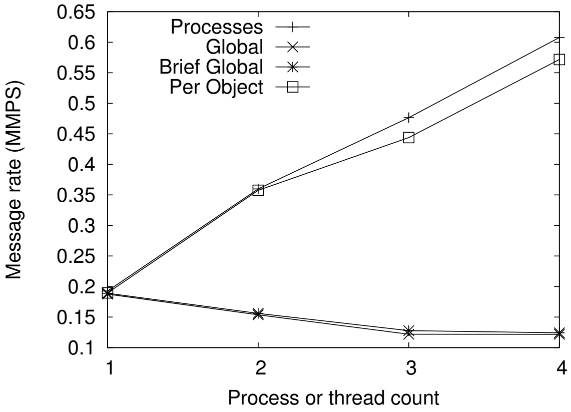


Fig. 4. Message rates with blocking sends for Global, Brief-Global, and Per-Object granularities.

4.2 Performance with Blocking Sends

This test measures the performance when the communication path is exercised, which requires critical sections to be acquired. The test measures the message rate for zero-byte blocking sends. (Even for zero-byte sends, the implementation must send the message envelope to the destination because the receives could have been posted for a larger size.)

Figure 4 shows the results. Notice that because of the cost of communication, the overall message rate is considerably lower than with `MPI_PROC_NULL`. In this test, even Brief Global performs as poorly as Global, because it acquires a large critical section during communication, which dominates the overall time. We then tried the Per-Object granularity, which demonstrated very good performance (comparable to multiple processes), because the granularity of critical sections in this case is per virtual channel (VC), rather than global. In MPICH2, a VC is a data structure that holds all the state and information required for a process to communicate with another process. Since each thread sends to a different process, they use separate VCs, and there is no contention for the critical section.

4.3 Performance with Nonblocking Sends

When performing a blocking send for short messages, MPICH2 does not need to allocate an `MPI_Request` object. For nonblocking sends, however, MPICH2 must allocate a request object to keep track of the progress of the communication operation. Requests are allocated from a common pool of free requests, which must be protected by a critical section. When a request is completed, it is freed and returned to the common pool. As a result, the common request pool becomes a source of critical-section contention.

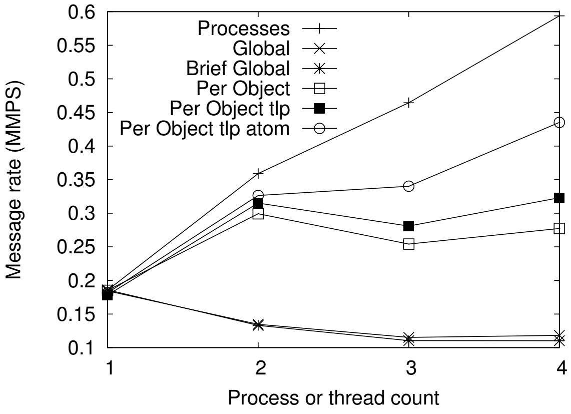


Fig. 5. Message rates with nonblocking sends. "Per Object tlp" is the thread-local request-pool optimization and "Per Object tlp atom" is the update of reference counts using atomic assembly instructions.

Each request object also uses a reference count to determine when the operation is complete and when it is safe to free the object. Since any thread can cause progress on communication, any thread can increment or decrement the reference count. A critical section is therefore needed, which can become another source of contention. All this makes it more difficult to minimize threading overhead in nonblocking sends than blocking sends.

We modified the test program to use nonblocking sends and measured the message rates. Figure 5 shows the results. Notice that the performance of Per-Object granularity is significantly affected by the contention on the request pool, and the message rate does not increase beyond more than two threads.

To reduce the contention on the common request pool, we experimented with providing a local free pool for each thread. These thread-local pools are initially empty. When a thread needs to allocate a request and its local pool is empty, it will get it from the common pool. But when a request is freed, it is returned to the thread's local pool. The next time the thread needs a request, it will allocate it from its local pool and avoid acquiring the critical section for the common request pool. The graph labeled "Per Object tlp" in Figure 5 shows that by adding the thread-local request pool, the message rate improves, but only slightly. The contention for the reference-count updates still hurts.

To alleviate the reference-count contention, we modified MPICH2 to use atomic assembly instructions for updating reference counts (instead of using a mutex). The graph labeled "Per Object tlp atom" in Figure 5 shows that the message rate improves even further with this optimization, and increases with the number of threads. It is still less than in the multiple-process case, but some performance degradation is to be expected with multithreading because critical sections cannot be completely avoided.

5 Conclusions and Future Work

We have studied the problem of improving the multithreaded performance of MPI implementations and presented several approaches to reduce the critical-section granularity, which can impact performance significantly. Such optimizations, however, require careful implementation.

While it is clear that atomic use and update of the communication engine is essential, it is equally important to ensure that all shared data structures, including MPI datatypes, requests, and communicators, are updated in a thread-safe way. For example, the reference-count updates used in most (if not all) MPI implementations must be thread atomic. This is not just a theoretical requirement: In some early experiments, we did not atomically update the reference counts, assuming that the very small race condition would not affect the results; but, by doing so, we regularly encountered failures in our communication-intensive tests. This experience suggests that the quasi-thread-safe approach proposed by Plachetka [11], in which only the access to the communication engine is serialized, is not sufficient.

The abstractions we have employed to control critical-section granularity are similar to what is required for transactional memory. We plan to use these abstractions to explore the use of transactional memory.

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. We thank Sameer Kumar and others in the MPI group at IBM Research and IBM Rochester for discussions about efficient support for thread safety in MPI.

References

1. Analysis of thread safety needs of MPI routines. `http://www.mcs.anl.gov/research/projects/mpich2/design/threadlist.htm`.
2. 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.
3. 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.
4. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.
5. 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.

6. William Gropp and Rajeev Thakur. Thread safety in an MPI implementation: Requirements and analysis. *Parallel Computing*, 33(9):595–604, September 2007.
7. Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.
8. Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.
9. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.
10. MPICH2. `http://www.mcs.anl.gov/mpi/mpich2`.
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. 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.
14. 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.
15. Meng-Shiou Wu, Srinivas Aluru, and Ricky A. Kendall. Mixed mode matrix multiplication. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2002)*, pages 195–203, September 2002.