Multithreading MPI Communication

Nicolae-Andrei Vasile

Faculty of Computer Science and Automatic Control

POLITEHNICA University of Bucharest

Bucharest, Romania

***Abstract* – As computing systems continue to grow in scale, recent advances in multi-core architectures have pushed such growth toward more denser architectures, that is, more processing elements per physical node, rather than more physical nodes themselves. Although a large number of scientific applications have relied so far on an MPI-everywhere model for programming high-end parallel systems, this model may not be sufficient for future machines, given their physical constraints such as decreasing amounts of memory per processing element and shared caches. As a result, application and computer scientists are exploring alternative programming models that involve using MPI between address spaces and some other threaded model, such as OpenMP or Pthreads. Such hybrid models require efficient support from an MPI implementation for MPI messages sent from multiple threads simultaneously.**

I. 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. In either case, MPI functions could be called from multiple threads of a process [1].

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 MPICH-MX. Other MPI implementations, such as MPICH, Open MPI, MVAPICH2, IBM MPI for Blue Gene/P and Power systems, and Intel, HP, SGI, and SUN MPIs do support thread safety [2], [3]. 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. In this paper, we describe our efforts at improving the multithreaded support in our MPI implementation, MPICH. 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.

Thread safety in MPI has been studied by a few researchers, but none of them have covered the topics discussed in this paper. Protopopov and Skjellum discuss a number of issues related to threads and MPI, including a design for a thread-safe version of MPICH-1 [4], [5]. 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 [6].

II. THREAD SAFETY IN MPI

For performance reasons, MPI defines four "levels" of thread safety [2], and allows the user to indicate the level desired—the idea being that the implementation needs to 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. In this paper we 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 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.

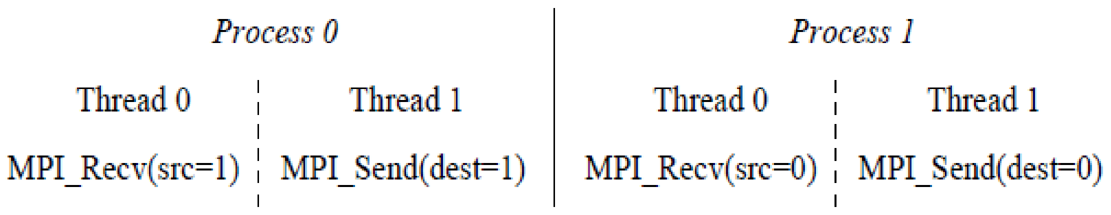


Figure 1. An implementation must ensure that this example never deadlocks for any ordering of thread execution

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.)

III. 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 [1], [2]. 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 [2]. 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 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. This is the simplest approach and is used in the past few releases of MPICH.

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.

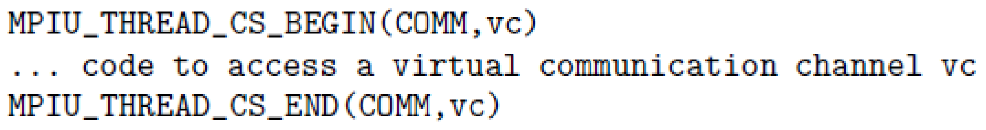
Per Object

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 [2], 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 MPICH, 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, please see Figure 2.



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

Figure 2. Passing a particular object to the critical section

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 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, one of the solutions proposed in reference [2], is to develop 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.

IV. 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 3 (a).

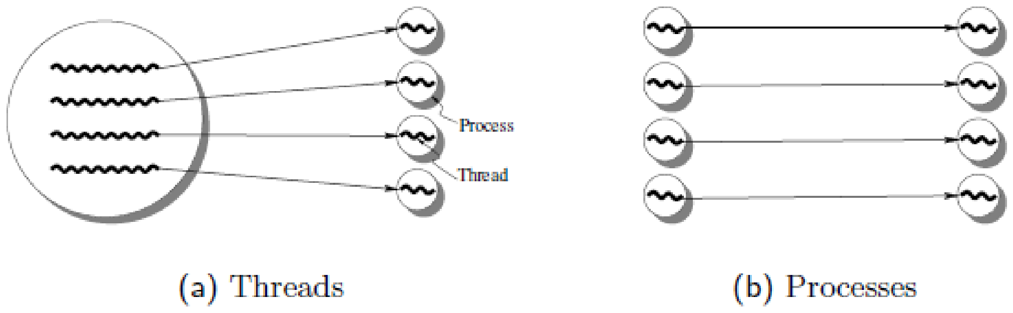


Figure 3. Illustration of test programs. Multiple threads or processes send messages to a different single-threaded receiving process.

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 3 (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.

Performance with *MPI_PROC_NULL*

This test is intended to measure the threading overhead in the MPICH 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 MPICH, 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.

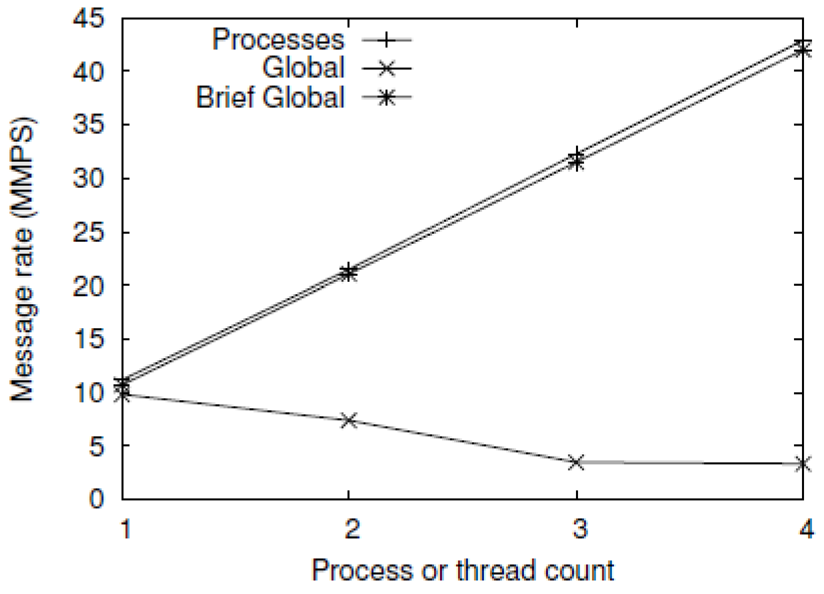


Figure 4. Message rate (in million messages per sec.) for a multithreaded process sending to MPI_PROC_NULL with Global and Brief Global granularities, compared to that with multiple processes.

Figure 4 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.

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).

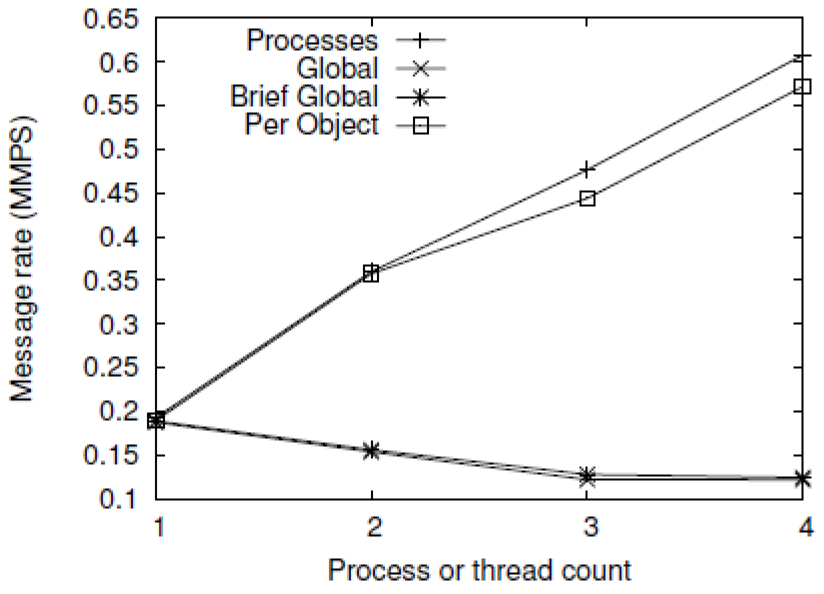


Figure 5. Message rates with blocking sends for Global, Brief-Global, and Per-Object granularities.

Figure 5 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 MPICH, 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.

Performance with Nonblocking Sends

When performing a blocking send for short messages, MPICH does not need to allocate an *MPI_Request* object. For nonblocking sends, however, MPICH 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.

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.

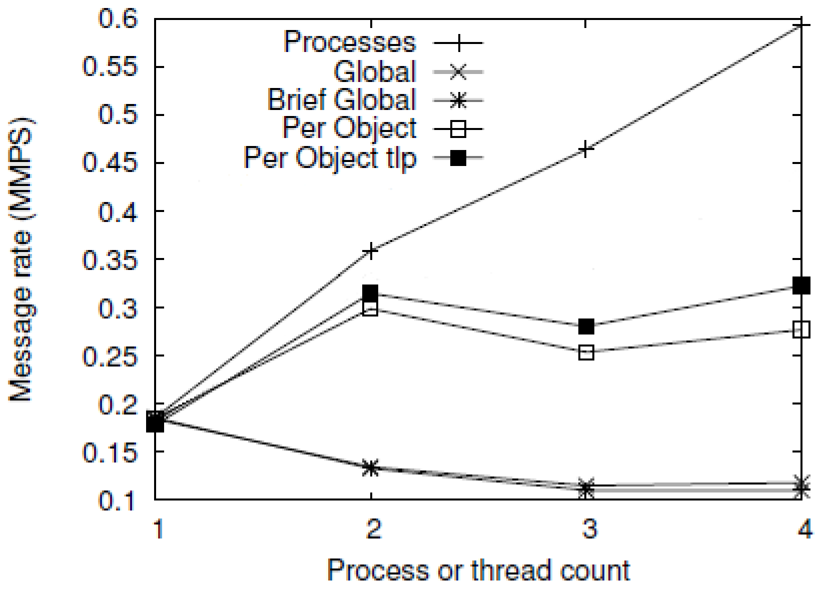


Figure 6. Message rates with nonblocking sends. "Per Object tlp" is the thread-local request-pool optimization.

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, reference [2] suggests using atomic assembly instructions for updating reference counts (instead of using a mutex). This way, 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.

V. CONCLUSIONS

In this paper, we have discussed several issues that must be considered when implementing thread safety in MPI. Some of the issues are subtle, but nonetheless important. Moreover, we have presented several approaches to reduce the critical-section granularity, which can impact performance significantly.

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. Implementing thread safety in MPI is not simple or straightforward. Careful thought and analysis are required in order to implement thread safety correctly and without sacrificing too much performance.

The default ch3:sock channel (TCP) in the 1.0.5 version of former MPICH2 is thread safe. The default build of the ch3:sock channel supports thread safety, but it is enabled only at run time if the user calls *MPI_Init_thread* with *MPI_THREAD_MULTIPLE*. If not, no thread locks are called, and so there is no penalty.

VI. REFERENCES

[1] R. Thakur and W. Gropp, "Test suite for evaluating performance of multithreaded MPI communication," *Parallel Computing,* vol. 35, no. 12, pp. 608-617, 2009.

[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp and R. Thakur, "Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming," *International Journal of High Performance Computing Applications,* vol. 24, no. 1, pp. 49-57, 2010.

[3] W. Gropp and R. Thakur, "Thread-safety in an MPI implementation: Requirements and analysis," *Parallel Computing,* vol. 33, no. 9, pp. 595-604, 2007.

[4] B. V. Protopopov and A. Skjellum, "A Multithreaded Message Passing Interface (MPI) Architecture," *Journal of Parallel and Distributed Computing,* vol. 61, no. 4, pp. 449-456, 2001.

[5] A. Skjellum, B. V. Protopopov and S. Hebert, "A Thread Taxonomy for MPI," in *Proceedings of the Second MPI Developers Conference*, Notre Dame, IN, USA, 1996.

[6] T. Plachetka, "(Quasi-) Thread-Safe PVM and (Quasi-) Thread-Safe MPI without Active Polling," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2002.