Thread Safety in an MPI Implementation: Requirements and Analysis

William Gropp and Rajeev Thakur *

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

Abstract

The MPI-2 Standard has carefully specified the interaction between MPI and user-created threads, with the goal of enabling users to write multithreaded programs while also enabling MPI implementations to deliver high performance. However, a simple reading of the thread-safety specification does not reveal what its implications are for an implementation and what implementers must be aware (and careful) of. In this paper, we describe and analyze what the MPI Standard says about thread safety and what it implies for an implementation. We classify the MPI functions based on their thread-safety requirements and discuss several issues to consider when implementing thread safety in MPI. We had to deal with many of these issues when designing and implementing thread safety in MPICH2. We use the example of generating new context ids (required for creating new communicators) to demonstrate how a simple solution for the single-threaded case cannot be used when there are multiple threads and how a naïve thread-safe algorithm can be expensive. We then present an algorithm for generating context ids that works efficiently in both single-threaded and multithreaded cases.

Key words: Message Passing Interface (MPI), thread safety, MPI implementation, multithreaded programming

1 Introduction

With SMP machines being commonly available and multicore chips becoming the norm, the mixing of the message-passing programming model with

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

multithreading on a single multicore chip or SMP node is gaining increasing attention. In such a mixed programming model, user programs consist of one or more MPI processes on each SMP node or multicore chip, with each MPI process itself comprising multiple threads. MPI implementations must be able to support such programs efficiently.

The MPI-2 Standard has clearly defined the interaction between MPI and user-created threads in an MPI program [7]. This specification was written with the goal of enabling users to write multithreaded MPI programs easily, without unduly burdening MPI implementations to support more than what a user might need. However, a simple reading of the Standard does not reveal all the implications the thread-safety specification has for an MPI implementation. Indeed, implementing thread safety in MPI correctly and without sacrificing too much performance requires careful thought and analysis.

In this paper, we discuss issues involved in developing an efficient thread-safe MPI implementation. We had to deal with many of these issues when designing and implementing thread safety in MPICH2 [8]. We first describe in brief the thread-safety specification in MPI. We then classify the MPI functions based on their thread-safety requirements. We discuss various issues to consider when implementing thread safety in MPI. In addition, we discuss the example of generating context ids and present an efficient, thread-safe algorithm for both single-threaded and multithreaded cases.

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 [11,12]. 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 [10]. García et al. present MiMPI, a thread-safe implementation of MPI [4]. TOMPI [3] and TMPI [13] are *thread-based* MPI implementations, where each MPI process is actually a thread. 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 [6].

2 What MPI Says about Thread Safety

The MPI-2 Standard [7] specifies the interaction between MPI calls and threads. MPI supports four "levels" of thread safety that a user must explicitly select:

`MPI_THREAD_SINGLE` A process has only one thread of execution.

`MPI_THREAD_FUNNELED` A process may be multithreaded, but only the thread that initialized MPI can make MPI calls.

`MPI_THREAD_SERIALIZED` A process may be multithreaded, but only one thread at a time can make MPI calls.

`MPI_THREAD_MULTIPLE` A process may be multithreaded and multiple threads can call MPI functions simultaneously.

The user must call the function `MPI_Init_thread` to indicate the level of thread-support desired, and the MPI implementation will return the level it supports. The user program must meet the restrictions of the level supported. The threads of a process are not separately addressable in MPI: A rank in a send or receive call identifies a process, not a thread. A message sent to a process may be received by any thread in that process that makes a matching receive call.

An implementation is not required to support levels higher than `MPI_THREAD_SINGLE`. In other words, an implementation is not required to be thread safe. A fully thread-compliant implementation, however, will support `MPI_THREAD_MULTIPLE`. A portable program that does not call `MPI_Init_thread` should assume that only `MPI_THREAD_SINGLE` is supported.

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.

Need for Multiple Levels of Thread Safety MPI requires the user to specify the level of thread safety needed because it comes at a cost. To demonstrate the cost of always using a thread-safe MPI implementation even when thread safety is not needed, we performed some experiments with MPICH2 (1.0.5). We measured the ping-pong (blocking send, blocking receive) latency between two single-threaded processes with MPICH2 configured in the following ways:

Single MPICH2 was configured with `--enable-threads=single`, which disables support for thread safety.

Runtime MPICH2 was configured with `--enable-threads=multiple`, which supports `MPI_THREAD_MULTIPLE`, and an additional runtime check was en-

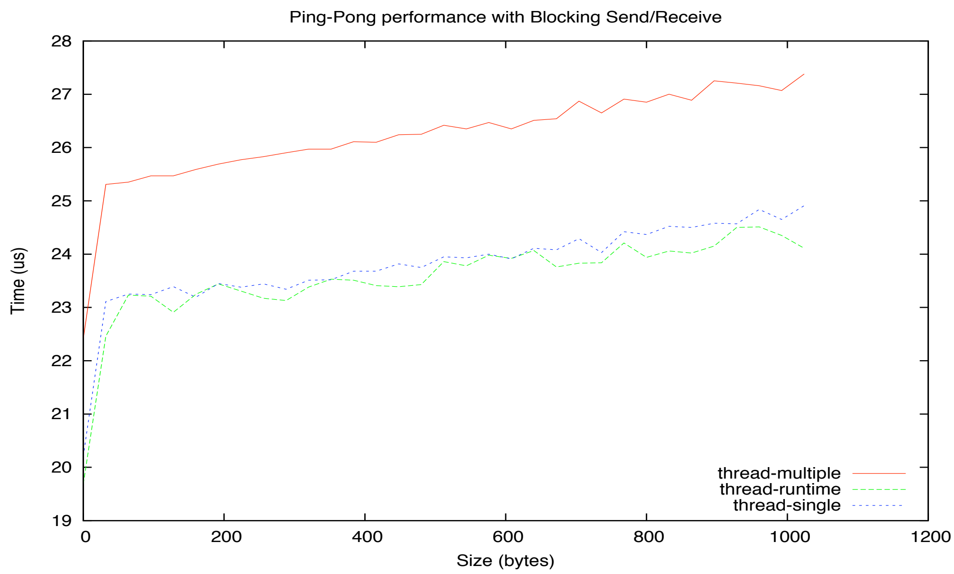

Fig. 1. Overhead of using a fully thread-safe MPI implementation when not needed.

abled that sets the default level to `MPI_THREAD_FUNNELED` (no thread locks) unless the user explicitly calls `MPI_Init_thread` requesting `MPI_THREAD_MULTIPLE`.

Multiple MPICH2 was configured with `--enable-threads=multiple`, and the default level was set to always be `MPI_THREAD_MULTIPLE`.

The tests were conducted on a single SMP box with the ch3:sock (TCP) channel in MPICH2 (currently the only channel in MPICH2 that supports thread safety). The results in Figure 1 show that the single and runtime cases perform about the same (within measurement error); that is, the runtime check for whether `MPI_THREAD_MULTIPLE` has been selected does not add overhead. The multiple case, however, is significantly more expensive even though there is only one thread. The cost of always acquiring and releasing thread locks (because of the need to assume that there may be multiple threads) adds significant overhead.

In the rest of this paper, we focus on the `MPI_THREAD_MULTIPLE` (fully multi-threaded) case.

3 Thread-Safety Classification of MPI Functions

We analyzed each MPI function (about 305 functions in all) to determine its thread-safety requirements. We then classified each function into one of several categories based on its primary requirement. The categories and examples of functions in those categories are described below; the complete classification can be found in [1].

None Either the function has no thread-safety issues, or the function has no thread-safety issues in correct programs and the function must have low overhead, so an optimized (nondebug) version need not check for race conditions. Examples: `MPI_Address`, `MPI_Wtick`.

Access Only The function accesses fixed data for an MPI object, such as the size of a communicator. This case differs from the "None" case because an erroneous MPI program could free the object in a race with a function that accesses the read-only data. A production MPI implementation need not guard this function against changes in another thread. This category may also include replacing a value in an object, such as setting the name of a communicator. Examples: `MPI_Comm_rank`, `MPI_Get_count`.

Update Ref The function updates the reference count of an MPI object. Such a function is typically used to return a reference to an existing object, such as a datatype or error handler. Examples: `MPI_Comm_group`, `MPI_File_get_view`.

Comm/IO The function needs to access the communication or I/O system in a thread-safe way. This is a very coarse-grained category but is sufficient to provide thread safety. In other words, an implementation may (and probably should) use finer-grained controls within this category. Examples: `MPI_Send`, `MPI_File_read`.

Collective The function is collective. MPI requires that the user not call collective functions on the same communicator in different threads in a way that may make the order of invocation depend on thread timing (race). Therefore, a production MPI implementation need not separately lock around the collective functions, but a debug version may want to detect races. The communication part of the collective function is assumed to be handled separately through the communication thread locks. Examples: `MPI_Bcast`, `MPI_Comm_spawn`.

Read List The function returns an element from a list of items, such as an attribute or info value. A correct MPI program will not contain any race that might update or delete the entry that is being read. This guarantee enables an implementation to use a lock-free, thread-safe set of list update and access operations in the production version; a debug version can attempt to detect improper race conditions. Examples: `MPI_Info_get`, `MPI_Comm_get_attr`.

Update List The function updates a list of items that may also be read. Multiple threads are allowed to simultaneously update the list, so the update implementation must be thread safe. Examples: `MPI_Info_set`, `MPI_Type_delete_attr`.

Allocate The function allocates an MPI object (may also need memory allocation such as with `malloc`). Examples: `MPI_Send_init`, `MPI_Keyval_create`.

Own The function has its own thread-safety management. Examples are "global" state such as buffers for `MPI_Bsend`. Examples: `MPI_Buffer_attach`, `MPI_Cart_create`.

Other Special cases. Examples: `MPI_Abort` and `MPI_Finalize`.

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

This classification helps an implementation determine the scope of the thread-safety requirements of various MPI functions and accordingly decide how to implement them. For example, functions that fall under the "None" or "Access Only" category need not have any thread lock in them. Appropriate thread locks can be added to other functions.

4 Issues in Implementing Thread Safety

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, which in turn might prevent the occurrence of the event that is needed for the blocked function to return. An example is shown in Figure 2. 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; this situation would cause the `MPI_Recv` to block forever.

In addition to using a more detailed strategy than simply locking around every function, an implementation must consider other issues that are described below. In particular, it is not enough to just lock around nonblocking communication calls and release the locks before calling a blocking communication call.

4.1 Updates of MPI Objects

A number of MPI objects, such as datatypes and communicators, have *reference-count* semantics. That is, the user can free a datatype after it has been used in a nonblocking communication operation even before that communication completes. MPI guarantees that the object will not be deleted until all uses have completed. A common way to implement this semantic is to maintain with each object a reference count that is incremented each time the object is used and decremented when the use is complete. In the multithreaded case, the reference count must be changed atomically because multiple threads could attempt to modify it simultaneously.

4.2 Thread-Private Memory

In the multithreaded case, an MPI implementation may sometimes need to use global or static variables that have different values on different threads. This cannot be achieved with regular variables because the threads of a process share a single memory space. Instead, one has to use special functions provided by the threads package for accessing thread-private memory (such as `pthread_getspecific`).

For example, thread-private memory is needed for keeping track of the "nesting level" of MPI functions. MPI functions may be nested because the implementation of an MPI function may call another MPI function. For example, the collective I/O functions may internally call MPI communication functions. If an error occurs in the nested MPI function, the implementation must not invoke the error handler. Instead, the error must be propagated back up to the top-level MPI function, and the error handler for that function must be invoked. This process requires keeping track of the nesting level of MPI functions and not invoking the error handler if the nesting level is more than one. (The implementation cannot simply reset the error handler before calling the nested function because the application may call the same function from another thread and expect the error handler to be invoked.) In the single-threaded case, an implementation could simply use a global variable to keep track of the nesting level; but in the multithreaded case, thread-private memory must be used.

Since accessing thread-private data requires a function call, implementations must ensure that such access is minimized in order to maintain good efficiency.

4.3 Memory Consistency

Updates to memory in one thread may not be seen in the same order by another thread. For example, some processors require an explicit *write barrier* to ensure that all memory-store operations have completed in memory. The lock and unlock operations for thread mutexes typically also perform the necessary synchronization operations needed for memory consistency. However, if an implementation avoids using mutex locks for higher performance and instead uses other mechanisms such as lock-free atomic updates, it must be careful to ensure that the memory updates happen as desired. This is a deep issue, a full discussion of which must include concepts such as sequential consistency and release consistency and is beyond the scope of this paper. Here, it suffices to say that an implementation must ensure that, for any object that multiple threads may access, the updates are consistent across all threads, not just the thread performing the updates.

4.4 Thread Failure

A major problem with any lock-based thread-safety model is what happens when a thread that holds a lock fails or is deliberately canceled (for example, with `pthread_cancel`). In that case, no other thread can acquire the lock, and the application may hang. One solution is to avoid using locks and instead use lock-free algorithms wherever possible (such as for the Update List category of functions described in Section 3).

4.5 Performance and Code Complexity

A tradeoff in performance and code complexity exists between using a single, coarse-grained lock and multiple, finer-grained locks. The single lock is relatively easy to implement but effectively serializes the MPI functions among threads. A finer-grained approach, using either multiple locks or a combination of locks and lock-free methods, risks the occurrence of deadly embrace (when two threads each hold one of the two locks that the other thread needs) as well as considerable code complexity. In addition, if the finer-grained approach requires multiple locks, it can be more expensive than if a single lock is used. MPI functions that can avoid using locks altogether by using lock-free methods (for example, with atomic test-and-set or compare-and-swap instructions) can provide a middle ground, trading a small amount of code complexity for more concurrency in execution.

4.6 Thread Scheduling

Another issue is avoiding "busy waiting" or "spin locks." In multithreaded code, it is common practice to have a thread that is waiting for an event (such as an incoming message for a blocking `MPI_Recv`) to yield to other threads, so that those threads can perform useful work. Thread systems provide various mechanisms for implementing this, such as condition variables. One difficulty is that not all events have the ability to wake up a thread; for example, if a low-latency method is being used to communicate between different processes in the same shared-memory node, there may be no easy way to signal the target process or thread. This situation often leads to a tradeoff between latency and effective scheduling.

5 An Algorithm for Generating Context Ids

In this section, we use the example of generating context ids (required for creating new communicators) to show how a simple solution for the single-threaded case cannot be used when there are multiple threads. We then present an efficient algorithm for generating context ids in the multithreaded case.

5.1 Basic Concept and Single-Threaded Solution

A communicator in MPI has a notion of a “context” associated with it, which is invisible to the user. This notion is implicit in a communicator and provides a safe communication space so that a message sent on a communicator is matched only by a receive posted on the same communicator (and not any other communicator).

Typically, the context is implemented as an integer that has the same value on all processes that are part of the communicator and is unique among all communicators on a given process. For example, if the context id of a communicator ‘X’ on a process is 42, all other processes that are part of X must use 42 as the context id for X, and no other communicator on any of these processes may use 42 as its context id. Processes that are not part of X, however, may use 42 as the context id for some other communicator.

Whenever a new communicator is created (for example, with `MPI_Comm_dup` or `MPI_Comm_create`), the processes in that communicator must agree on a context id for the new communicator, following the constraints given above. In the single-threaded case, generating a new context id is easy. One approach could be for each process to maintain a global data structure containing the list of available context ids on that process. In order to save memory space, the list can be maintained as a bit vector, with the bits indicating whether the corresponding context ids are available. A new context id can be generated by performing an `MPI_Allreduce` with the appropriate bit operator (`MPI_BAND`). The position of the lowest set bit can be used as the new context id.

5.2 Naïve Multithreaded Algorithm

The multithreaded case is more difficult. A process cannot simply acquire a thread lock, call `MPI_Allreduce`, and release the lock, because the threads on various processes may acquire locks in different order, causing the allreduce operation to hang because of a deadly embrace.

One possible solution is to acquire a thread lock, read the bit vector, release the lock, then do the `MPI_Allreduce`, followed by another `MPI_Allreduce` to determine whether the bit vector has been changed by another thread between the lock release and the first allreduce. If not, then the value for the context id can be accepted; otherwise, the algorithm must be repeated. This method is expensive, however, as it requires multiple `MPI_Allreduce` calls. In addition, two competing threads could loop forever, with each thread invalidating the other's choice of context value.

5.3 Efficient Algorithm for the Multithreaded Case

We instead present a new algorithm that works efficiently in both single-threaded and multithreaded cases. We have implemented this algorithm in MPICH2 [8]. For simplicity, we present the algorithm only for the case of intracommunicators. The pseudocode is given in Figure 3.

The algorithm uses a bit mask of context ids; each bit set indicates a context id available. For example, 32 32-bit integers will cover 1024 context ids. This mask and two other variables, `lowestContextId` and `mask_in_use`, are stored in global memory (shared among the threads of a process). The variable `lowestContextId` is used to store the smallest context id among the input communicators of the various threads on a process that need to find a new context id. The variable `mask_in_use` indicates whether some thread has acquired the rights to the mask.

The algorithm works as follows. A thread wishing to get a new context id first acquires a thread lock. If `mask_in_use` is set or the context id of the thread's input communicator is greater than `lowestContextId`, the thread uses 0 as the `local_mask` (for allreduce) and sets the flag `i_own_the_mask` to 0. Otherwise, it uses the current context-id mask as the `local_mask` (for allreduce) and sets the flags `mask_in_use` and `i_own_the_mask` to 1. Then it releases the lock and does an `MPI_Allreduce` on `local_mask`. This operation is collective over the input communicator passed to the thread.

After `MPI_Allreduce` returns, if `i_own_the_mask` is 1, the thread acquires the lock again. If the result of the allreduce (`local_mask`) is not 0, it means all

```
/* global variables (shared among threads of a process) */
mask          /* bit mask of context ids in use by a process */
mask_in_use        /* flag; initialized to 0 */
lowestContextId    /* initialized to MAXINT */

/* local variables (not shared among threads) */
local_mask      /* local copy of mask */
i_own_the_mask /* flag */
context_id      /* new context id; initialized to 0 */

while (context_id == 0) {
    Mutex_lock()
    if (mask_in_use || MyComm->contextid > lowestContextId) {
        local_mask = 0
        i_own_the_mask = 0
        if (MyComm->contextid < lowestContextId) {
            lowestContextId = MyComm->contextid
        }
    }
    else {
        local_mask = mask
        mask_in_use = 1
        i_own_the_mask = 1
        lowestContextId = MyComm->contextid
    }
    Mutex_unlock()

    MPI_Allreduce(local_mask, MPI_BAND, MyComm)

    if (i_own_the_mask) {
        Mutex_lock()
        if (local_mask != 0) {
            context_id =
                location of first set bit in local_mask
            update mask
            if (lowestContextId == MyComm->contextid) {
                lowestContextId = MAXINT;
            }
        }
        mask_in_use = 0
        Mutex_unlock()
    }
}
return context_id
```

Fig. 3. Pseudocode for generating a new context id in the multithreaded case (for intracommunicators).

threads that participated in the allreduce owned the mask on their processes and therefore the location of the first set bit in `local_mask` can be used as the new context id. If the result of the allreduce is 0, it means that some thread did not own the mask on its process and therefore the algorithm must be retried. The variable `mask_in_use` is reset to 0 before releasing the lock.

The logic for `lowestContextId` exists to prevent a livelock situation where the allreduce operation always contains some threads that do not own the mask, resulting in a 0 output. Since threads in our algorithm yield ownership of the mask to the thread with the lowest context id, there will be a time when all the threads of the communicator with the lowest context id will own the mask on their respective processes, causing the allreduce to return a nonzero result, and a new context id to be found. Those threads will disappear from the contention, and the same algorithm will enable other threads to complete their operation.

In this algorithm, the case where different threads of a process may have the same input context id does not arise because it is not legal for multiple threads of a process to call collective functions with the same communicator at the same time, and all the MPI functions that need to create new context ids (namely, the functions that return new communicators) are collective functions.

We note that, in the single-threaded case, this algorithm is as efficient as the basic algorithm described in Section 5.1, because the mutex locks can be commented out and no extra communication is needed as the first allreduce itself will succeed. Even in the multithreaded case, in most common circumstances, the first allreduce will succeed, and no extra communication will be needed.

5.3.1 Correctness

Although we do not have a formal proof for the correctness of the algorithm, we have implemented it in MPICH2 and tested it extensively. In addition, one of our collaborators has tested the algorithm using formal verification techniques—by writing a formal model for the algorithm in Promela and verifying it with the SPIN [5] model checker. He was not able to find any bugs, deadlocks, or livelocks [9].

5.3.2 Performance

To study the performance of this algorithm with respect to the basic single-threaded algorithm described in Section 5.1, we ran three experiments to measure the performance of the MPI function `MPI_Comm_dup`, in which the most time-consuming operation is the generation of a new context id:

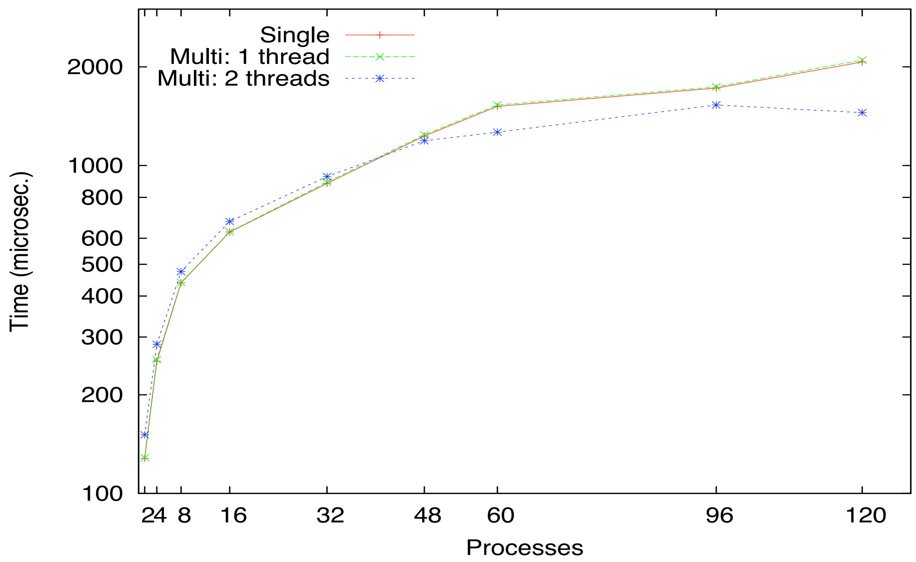

Fig. 4. Performance of the context id algorithm.

Single Using the single-threaded algorithm
Multi1 Using the multithreaded algorithm but each process has only one thread
Multi2 Using the multithreaded algorithm with each process having two threads, both calling `MPI_Comm_dup` (on different communicators)

In all cases, we called `MPI_Comm_dup` several times in a loop and measured the average time for a single call. The experiments were run on a Myrinet-connected Linux cluster using the ch3:sock channel (TCP) in MPICH2 1.0.5 (the only MPICH2 channel that currently supports thread safety). The results are shown in Figure 4.

The difference between the single and multi1 cases shows that the overhead of the multithreaded algorithm over the single-threaded case is negligible (the two lines almost overlap). For small numbers of processes, the multi2 case is only slightly more expensive than the single-threaded and multi1 cases because of contention between the two threads for locks and resources. For larger numbers of processes, however, the multi2 case in fact outperforms single and multi1. The reason is that when one thread waits for communication, some of that time is used by the other thread for its own `MPI_Comm_dup`. That is, the latency cost gets overlapped (when one thread blocks, it does not block the entire process). The results also indicate that the multithreaded algorithm does not require any more communication than does the single-threaded algorithm.

5.3.3 Further Improvements

A refinement to this algorithm could be to allow multiple threads to have disjoint masks; if the masks are cleverly picked, most threads would find an acceptable value even if multiple threads were concurrently executing the algorithm. Another refinement could be to use a queue of pending threads ordered by increasing context id of the input communicator. Threads that are high in this queue could wait on a condition variable or other thread-synchronization mechanism that is activated whenever there is a change in the thread with the lowest context id, either because a thread has found a new context id and is removed from the queue or because a new thread with a lower context id enters the function.

6 Conclusions and Future Work

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. In this paper, we have discussed several issues that an implementation must consider when implementing thread safety in MPI. Some of the issues are subtle, but nonetheless important.

The default ch3:sock channel (TCP) in the current version of MPICH2 (1.0.5) 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. We are working on performance improvements to the thread support in MPICH2 and extending thread safety to all the communication channels.

Although many MPI implementations claim to be thread safe, no comprehensive test suite exists to validate the claim. We plan to develop a test suite that can be used to verify the thread safety of MPI implementations.

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. Dept. of Energy, under Contract DE-AC02-06CH11357.

References

[1] Analysis of thread safety needs of MPI routines. http://www.mcs.anl.gov/mpi/mpich2/developer/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] 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.

[5] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

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

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

[8] MPICH2. http://www.mcs.anl.gov/mpi/mpich2.

[9] Salman Pervez. Personal communication, 2006.

[10] 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.

[11] 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.

[12] 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.

[13] 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.