Optimizing the Synchronization Operations in MPI One-Sided Communication*

Rajeev Thakur   *William Gropp*   *Brian Toonen*
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439, USA

{thakur, gropp, toonen}@mcs.anl.gov

Abstract

One-sided communication in MPI requires the use of one of three different synchronization mechanisms, which indicate when the one-sided operation can be started and when the operation is completed. Efficient implementation of the synchronization mechanisms is critical to achieving good performance with one-sided communication. Our performance measurements, however, indicate that in many MPI implementations, the synchronization functions add significant overhead, resulting in one-sided communication performing much worse than point-to-point communication for short- and medium-sized messages. In this paper, we describe our efforts to minimize the overhead of synchronization in our implementation of one-sided communication in MPICH2. We describe our optimizations for all three synchronization mechanisms defined in MPI: fence, post-start-complete-wait, and lock-unlock. Our performance results demonstrate that, for short messages, MPICH2 performs six times faster than LAM for fence synchronization and 50% faster for post-start-complete-wait synchronization, and it performs more than twice as fast as Sun MPI for all three synchronization methods.

*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 W-31-109-ENG-38.

1 Introduction

MPI defines one-sided communication operations that allow users to directly read from or write to the memory of a remote process [13]. One-sided communication both is convenient to use and has the potential to deliver higher performance than regular point-to-point (two-sided) communication, particularly on networks that support one-sided communication natively, such as InfiniBand [9] and Myrinet [16]. On networks that support only two-sided communication, such as TCP, it is harder for one-sided communication to do better than point-to-point communication. Nonetheless, a good implementation should strive to deliver performance as close as possible to that of point-to-point communication.

One-sided communication in MPI requires the use of one of three synchronization mechanisms: fence, post-start-complete-wait, or lock-unlock. The synchronization mechanism defines the time at which the user can initiate one-sided communication and the time when the operations are guaranteed to be completed. The true cost of one-sided communication, therefore, must include the time taken for synchronization. An unoptimized implementation of the synchronization functions may perform more communication and synchronization than necessary (such as a barrier), which can adversely affect performance, particularly for short and medium-sized messages.

To determine how the different synchronization mechanisms perform in existing MPI implementations, we used a test program that performs nearest-neighbor ghost-area (or halo) exchange, a communication pattern common in many scientific applications such as PDE simulations. We wrote four versions of this program: using point-to-point communication (isend/irecv) and using one-sided communication with fence, post-start-complete-wait, and lock-unlock synchronization. We measured the time taken for a single communication step (each process exchanges data with its four neighbors) by doing the step a number of times and calculating the average. Figure 1 shows a template of the fence version of the test; the other versions are similar.

We ran the tests with three different MPI implementations: IBM MPI, Sun MPI, and LAM [12]. For IBM MPI, we used the IBM SP (Blue Horizon) at San Diego Supercomputer Center; for Sun MPI, we used a Sun Fire E6900 SMP at the University of Aachen, Germany; and for LAM, we used a fast-ethernet-connected Linux cluster at Argonne. The performance results are shown in Figure 2. With IBM MPI on an SP, one-sided communication is almost two orders of magnitude slower than point-to-point (pt2pt) for short messages and remains significantly slower until messages get larger than 256 KB. With Sun MPI on a shared-memory SMP, all three one-sided versions are about six times slower than the point-to-point version for short messages. With LAM

```
for (i=0; i<ntimes; i++) {
   MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
   for (j=0; j<nbrs; j++) {
       MPI_Put(sbuf + j*n, n, MPI_INT, nbr[j], j, n, MPI_INT, win);
   }
   MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT |
                 MPI_MODE_NOSUCCEED, win);
}
```

Figure 1: Fence version of the test

on a Linux cluster connected with fast ethernet, for short messages, post-start-complete-wait (pscw) is about three times slower than point-to-point, and fence is about 18 times slower than point-to-point.[1] As shown in Figure 1, we pass appropriate assert values to `MPI_Win_fence` so that the MPI implementation can optimize the function. Since LAM does not support asserts, we commented them out when using LAM. At least some of the poor performance of LAM with fence can be attributed to not taking advantage of asserts.

We observed similar results for runs with different numbers of processes on all three implementations. Clearly, the overhead associated with synchronization significantly affects the performance of these implementations. Other researchers [6] have found similarly high overheads in their experiments with four MPI implementations: NEC, Hitachi, Sun, and LAM.

Our goal in the design and implementation of one-sided communication in our MPI implementation, MPICH2, has been to minimize the amount of additional communication and synchronization needed to implement the semantics defined by the synchronization functions. We particularly avoid using a barrier anywhere. As a result, we are able to achieve much higher performance than do other MPI implementations. We describe our optimizations and our implementation in this paper. This paper is an extended version of the work described in [18].

The rest of the paper is organized as follows. In Section 2, we describe related work in the area of one-sided communication. In Section 3, we explain the semantics of the three synchronization mechanisms. In Section 4, we describe our optimized implementation of the synchronization mechanism. In Section 5, we present performance results. In Section 6, we conclude with a brief discussion of future work.

[1] LAM does not support lock-unlock synchronization.

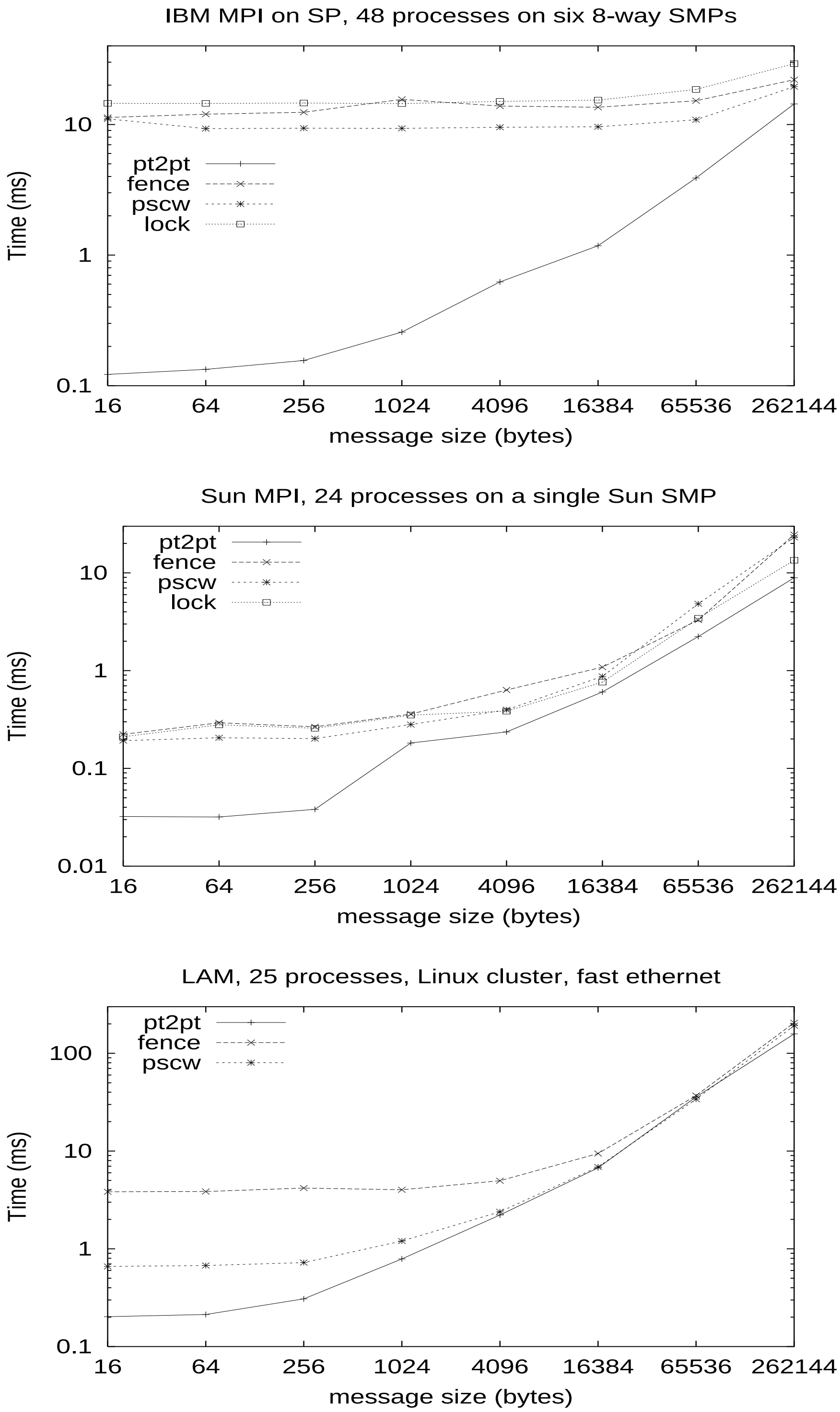


Figure 2: Performance of IBM MPI, Sun MPI, and LAM for a nearest-neighbor ghost-area exchange test

2 Related Work

One-sided communication as a programming paradigm was made popular initially by the SHMEM library on the Cray T3D and T3E [8], the BSP library [7], and the Global Arrays library [17]. After the MPI-2 Forum defined an interface for one-sided communication in MPI, several vendors and a few research groups implemented it, but, as far as we know, none of these implementations specifically optimizes the synchronization overhead. For example, the implementations of one-sided communication for Sun MPI by Booth and Mourão [4] and for the NEC SX-5 by Träff et al. [19] use a barrier to implement fence synchronization. Other efforts at implementing MPI one-sided communication include the implementation for InfiniBand networks by Jiang et al. [10, 11], for a Windows implementation of MPI (WMPI) by Mourão and Silva [15], for the Fujitsu VPP5000 vector machine by Asai et al. [1], and for the SCI interconnect by Worringen et al. [21]. Mourão and Booth [14] describe issues in implementing one-sided communication in an MPI implementation that uses multiple protocols, such as TCP and shared memory.

3 One-Sided Communication in MPI

In MPI, the memory that a process allows other processes to access via one-sided communication is called a *window*. Processes specify their local windows to other processes by calling the collective function `MPI_Win_create`. The three functions for one-sided communication are `MPI_Put`, `MPI_Get`, and `MPI_Accumulate`. `MPI_Put` writes to remote memory, `MPI_Get` reads from remote memory, and `MPI_Accumulate` does a reduction operation on remote memory. All three functions are nonblocking: They initiate but do not necessarily complete the one-sided operation. These functions are not sufficient by themselves because one needs to know when a one-sided operation can be initiated (that is, when the remote memory is ready to be read or written) and when a one-sided operation is guaranteed to be completed. To specify these semantics, MPI defines three different synchronization mechanisms.

3.1 Fence Synchronization

Figure 3a illustrates the fence method of synchronization (without the syntax). `MPI_Win_fence` is collective over the communicator associated with the window object. A process may issue one-sided operations after the first call to `MPI_Win_fence` returns. The next fence completes any one-sided operations that this process issued after the preceding fence, as well as the one-sided operations other processes issued that had this process as the target. The drawback of the fence

```
Process 0                 Process 1
MPI_Win_fence(win)        MPI_Win_fence(win)
MPI_Put(1)                MPI_Put(0)
MPI_Get(1)                MPI_Get(0)
MPI_Win_fence(win)        MPI_Win_fence(win)
```

a. Fence synchronization

```
Process 0               Process 1               Process 2
                        MPI_Win_post(0,2)
MPI_Win_start(1)                                MPI_Win_start(1)
MPI_Put(1)                                      MPI_Put(1)
MPI_Get(1)                                      MPI_Get(1)
MPI_Win_complete(1)                             MPI_Win_complete(1)
                        MPI_Win_wait(0,2)
```

b. Post-start-complete-wait synchronization

```
Process 0                Process 1             Process 2
MPI_Win_create(&win)     MPI_Win_create(&win)  MPI_Win_create(&win)
MPI_Win_lock(shared,1)                         MPI_Win_lock(shared,1)
MPI_Put(1)                                     MPI_Put(1)
MPI_Get(1)                                     MPI_Get(1)
MPI_Win_unlock(1)                              MPI_Win_unlock(1)
MPI_Win_free(&win)       MPI_Win_free(&win)    MPI_Win_free(&win)
```

c. Lock-unlock synchronization

Figure 3: The three synchronization mechanisms for one-sided communication in MPI. The numerical arguments indicate the target rank.

method is that if only small subsets of processes are actually communicating with each other, such as groups of nearest neighbors in our ghost-area exchange test program, the collectiveness of the fence function over the entire communicator results in unnecessary synchronization overhead.

3.2 Post-Start-Complete-Wait Synchronization

To avoid the drawback of fence, MPI defines a second synchronization method in which only subsets of processes need to synchronize, as shown in Figure 3b. A process that wishes to expose its local window to remote accesses calls `MPI_Win_post`, which takes as argument an `MPI_Group` object that specifies the set of processes that will access the window. A process that wishes to perform one-sided communication calls `MPI_Win_start`, which also takes as argument an `MPI_Group` object that specifies the set of processes that will be the target of one-sided operations from this process. After issuing all the one-sided operations, the origin process calls `MPI_Win_complete` to complete the operations at the origin. The target calls `MPI_Win_wait` to complete the operations at the target.

3.3 Lock-Unlock Synchronization

In the lock-unlock synchronization method, the origin process calls `MPI_Win_lock` to obtain either shared or exclusive access to the window on the target, as shown in Figure 3c. `MPI_Win_lock` is not required to block until the lock is acquired, except when the origin and target are one and the same process. After issuing the one-sided operations, the origin calls `MPI_Win_unlock`. When `MPI_Win_unlock` returns, the one-sided operations are guaranteed to be completed at the origin and the target. The target process does not make any synchronization call. For this synchronization method, an implementation is allowed to restrict the use of one-sided communication to windows in memory that have been allocated with the function `MPI_Alloc_mem`.

4 Implementing MPI One-Sided Communication

We describe our implementation of one-sided communication in MPICH2, particularly how we have optimized the implementation of the synchronization functions. Our current implementation of one-sided communication is layered on the same lower-level communication abstraction we use for point-to-point communication, called CH3 [2]. CH3 uses a two-sided communication model in which the sending side sends packets, followed optionally by data, and the receiving side explicitly posts receives for packets and, optionally, data. The content and interpretation of the packets are decided by the upper layers. To implement one-sided communication, we have simply added new packet types. So far, CH3 has been implemented robustly on top of TCP and shared memory, and experimental implementations exist for GASNet [3] and InfiniBand. Therefore, our implementation of one-sided communication runs on all these devices.

We use the following general approach in implementing the synchronization functions: For all three synchronization methods, we do almost nothing in the first synchronization call; do nothing in the calls to put, get, or accumulate other than queuing up the requests locally; and instead do everything in the second synchronization call. This approach allows the first synchronization call to return immediately without blocking, reduces or eliminates the need for extra communication in the second synchronization call, and offers the potential for communication operations to be aggregated and scheduled efficiently as in BSP [7]. We describe our implementation in detail below.

4.1 Fence Synchronization

An implementation of fence synchronization must take into account the following semantics: A one-sided operation cannot access a process's window until that process has called fence, and the next fence on a process cannot return until all processes that need to access that process's window have completed doing so.

A naïve implementation of fence synchronization could be as follows. At the first fence, all processes do a barrier so that everyone knows that everyone else has called fence. Puts, gets, and accumulates can be implemented as either blocking or nonblocking operations. In the second fence, after all the one-sided operations have been completed, all processes again do a barrier to ensure that no process leaves the fence before other processes have finished accessing its window. This method requires two barriers, which can be quite expensive.

In our implementation, we avoid the two barriers completely. In the first call to fence, we do nothing. For the puts, gets, and accumulates that follow, we simply queue them up locally and do nothing else, with the exception that any one-sided operation whose target is the origin process itself is performed immediately by doing a simple memory copy or local accumulate. In the second fence, each process goes through its list of queued one-sided operations and determines, for every other process i, whether any of the one-sided operations have i as the target. This information is stored in an array, such that a 1 in the ith location of the array means that one or more one-sided operations are targeted to process i, and a 0 means no one-sided operations are targeted to that process. Figure 4 shows an example. All processes now do a reduce-scatter sum operation on this array: Process i gets the result of the summation of the ith element of the array on all processes, as in `MPI_Reduce_scatter`. As a result, each process now knows how many processes will be performing one-sided operations on its window, and this number is stored in a counter in the `MPI_Win` object. Each process is now free to perform the data transfer for its one-sided operations; it needs only to ensure that the window counter at the target gets decremented when all the one-sided operations from this process to that target have been completed.

A put is performed by sending a put packet containing the address, count, and datatype information for the target. If the datatype is a derived datatype, an encapsulated version of the

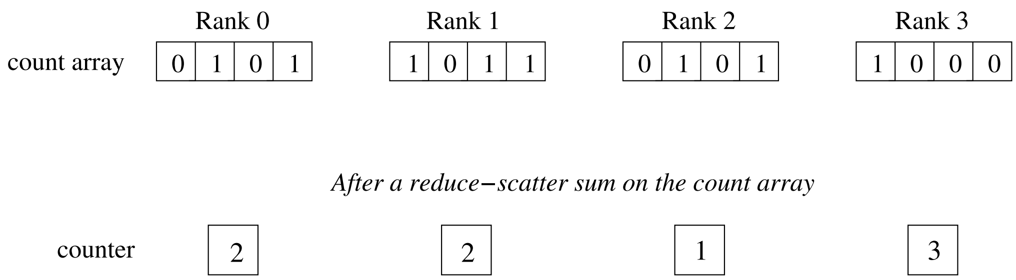


Figure 4: An example illustrating the use of reduce-scatter in the implementation of fence synchronization. On each process, a 1 in the ith location of the count array means that one or more one-sided operations are targeted to process i.

derived datatype is sent next. Then follows the actual data. The MPI progress engine on the target receives the packet and derived datatype, if any, and then directly receives the data into the correct memory locations. No rendezvous protocol is needed for the data transfer because the origin has already been authorized to write to the target window. Gets and accumulates are implemented similarly.

For the last one-sided operation, the origin process sets a field in the packet header indicating that it is the last operation. The target therefore knows to decrement its window counter after this operation (and all previous operations from that origin process) have completed at the target. When the counter reaches 0, it indicates that all remote processes that need to access the target's window have completed their operations, and the target can therefore return from the second fence.

We have thus eliminated the need for a barrier in the first fence and replaced the barrier at the *end* of the second fence by a reduce-scatter at the *beginning* of the second fence before any data transfer. After that, all processes can do their communication independently and return when they are done.

4.2 Post-Start-Complete-Wait Synchronization

An implementation of post-start-complete-wait synchronization must take into account the following semantics: A one-sided operation cannot access a process's window until that process has called `MPI_Win_post`, and a process cannot return from `MPI_Win_wait` until all processes that need to access that process's window have completed doing so and called `MPI_Win_complete`.

A naïve implementation of this synchronization could be as follows. `MPI_Win_start` blocks until it receives a message from all processes in the target group indicating that they have called MPI_Win_post. Puts, gets, and accumulates can be implemented as either blocking or nonblocking functions. MPI_Win_complete waits until all one-sided operations initiated by that process have completed locally and then sends a done message to each target process. MPI_Win_wait on the target blocks until it receives the done message from each origin process. Clearly, this method involves a great deal of synchronization.

We have eliminated most of this synchronization in our implementation as follows. In MPI_Win_post, if the assert MPI_MODE_NOCHECK is not specified, the process sends a zero-byte message to each process in the origin group to indicate that MPI_Win_post has been called. It also sets the counter in the window object to the size of this group. As in the fence case, this counter will get decremented by the completion of the last one-sided operation from each origin process. MPI_Win_wait simply blocks and invokes the progress engine until this counter reaches zero.

On the origin side, we do nothing in MPI_Win_start. All the one-sided operations following MPI_Win_start are simply queued up locally as in the fence case. In MPI_Win_complete, the process first waits to receive the zero-byte messages from the processes in the target group. It then performs all the one-sided operations exactly as in the fence case. The last one-sided operation has a field set in its packet that causes the target to decrement its counter on completion of the operation. If an origin process has no one-sided operations destined to a target that was part of the group passed to MPI_Win_start, it still needs to send a packet to that target for decrementing the target's counter. MPI_Win_complete returns when all its operations have locally completed.

Thus the only synchronization in this implementation is the wait at the beginning of MPI_Win_complete for a zero-byte message from the processes in the target group, and this too can be eliminated if the user specifies the assert MPI_MODE_NOCHECK to MPI_Win_post and MPI_Win_start (similar to MPI_Rsend).

4.3 Lock-Unlock Synchronization

Implementing lock-unlock synchronization when the window memory is not directly accessible by all origin processes requires the use of an asynchronous agent at the target to cause progress to occur, because one cannot assume that the user program at the target will call any MPI functions that will cause progress periodically. Our design for the implementation of lock-unlock synchronization involves the use of a thread that periodically wakes up and invokes the MPI progress engine if it finds that no other MPI function has invoked the progress engine within

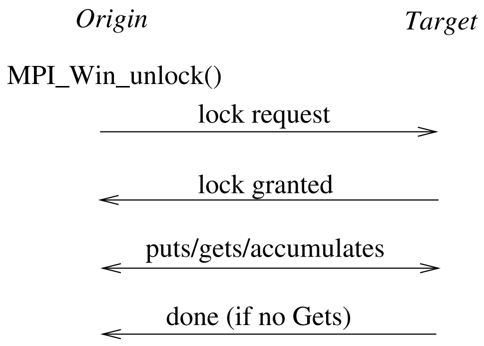


Figure 5: Implementing lock-unlock synchronization when there are multiple one-sided operations

some time interval.[2] If the progress engine had been invoked by other calls to MPI, the thread does nothing. This thread is created only when `MPI_Win_create` is called and if the user did not pass an info object to `MPI_Win_create` with the key `no_locks` set to `true` (indicating that he will not be using lock-unlock synchronization).

We implement lock-unlock synchronization as follows. In `MPI_Win_lock`, we do nothing but queue up the lock request locally and return immediately. The one-sided operations are also queued up locally. All the work is done in `MPI_Win_unlock`. For the general case where there are multiple one-sided operations, we implement `MPI_Win_unlock` as follows (see Figure 5). The origin sends a "lock-request" packet to the target and waits for a "lock-granted" reply. When the target receives the lock request, it either grants the lock by sending a lock-granted reply to the origin or queues up the lock request if it conflicts with the existing lock on the window. When the origin receives the lock-granted reply, it performs the one-sided operations exactly as in the other synchronization modes. The last one-sided operation, indicated by a field in the packet header, causes the target to release the lock on the window after the operation has completed. Therefore, no separate unlock request needs to be sent from origin to target.

The semantics specify that `MPI_Win_unlock` cannot return until the one-sided operations are completed at both origin and target. Therefore, if the lock is a shared lock and none of the operations is a get, the target sends an acknowledgment to the origin after the last operation has completed. If any one of the operations is a get, we reorder the operations and perform the get last. Since the origin must wait to receive data, no additional acknowledgment is needed. This

[2] Since the MPICH2 progress engine is not yet fully thread safe, we do not use this separate thread in the implementation yet. The main thread on the target must make calls to MPI functions for progress to occur. For example, the target could call `MPI_Win_free`, which will block until all one-sided communication has completed.

approach assumes that data transfer in the network is ordered. If not, an acknowledgment is needed even if the last operation is a get. If the lock is an exclusive lock, no acknowledgment is needed even if none of the operations is a get, because the exclusive lock on the window prevents another process from accessing the data before the operations have completed.

4.3.1 Optimization for Single Put/Get/Accumulate Operations

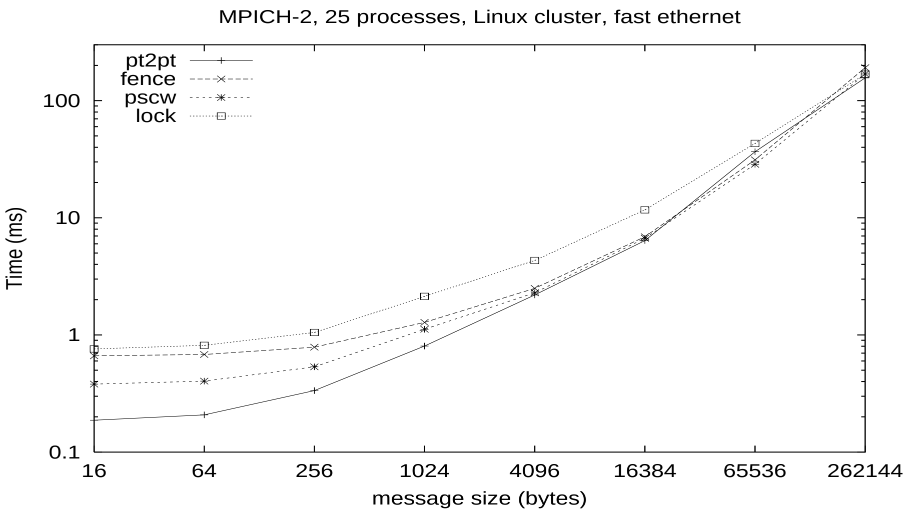
If the lock-unlock is for a single short operation and predefined datatype at the target, we send the put/accumulate data or get information along with the lock-request packet itself. If the target can grant the lock, it performs the specified operation right away. If not, it queues up the lock request along with the data or information and performs the operation when the lock can be granted. Except in the case of get operations, `MPI_Win_unlock` blocks until it receives an acknowledgment from the target that the operation has completed. This acknowledgment is needed even if the lock is an exclusive lock because the origin does not know whether the lock has been granted. Similar optimizations are possible for multiple one-sided operations, but at the cost of additional queuing/buffering at the target.

5 Performance Results

We studied the performance of our implementation by using the same ghost-area exchange program described in Section 1. Figure 6 shows the performance of the test program with MPICH2 on the fast-ethernet-connected Linux cluster and on the Sun SMP using shared memory. We see that the time taken by the point-to-point version with MPICH2 is about the same as with other MPI implementations in Figure 2, but the time taken by the one-sided versions is much lower. To compare the performance of MPICH2 with other MPI implementations, we calculated the ratio of the time with point-to-point communication to the time with one-sided communication and tabulated the results in Tables 1 and 2.[3] For short messages, MPICH2 is almost six times faster than LAM for the fence version and about 50% faster for the post-start-complete-wait version. Compared with Sun MPI for short messages, MPICH2 is more than twice as fast for all three synchronization methods. The difference narrows for large message sizes where the synchronization overheads are less of an issue, but MPICH2 still performs better than both LAM and Sun MPI.

We note that although we have succeeded in minimizing the synchronization overhead as-

[3] Since MPICH2 does not run on an IBM SP yet, we could not compare with IBM MPI.



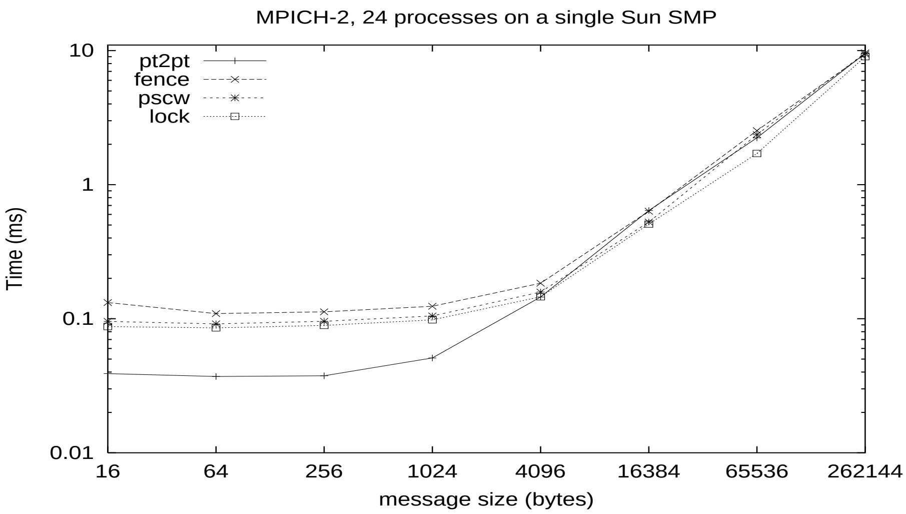


Figure 6: Performance of MPICH2 on the Linux cluster and Sun SMP

sociated with one-sided communication, MPI point-to-point communication still outperforms one-sided communication in our experiments. We attribute this to the following reasons:

- Since we use the CH3 communication layer in MPICH2 as the basis for our implementation of one-sided communication, and CH3 uses a two-sided communication model, MPI one-sided communication effectively gets converted to two-sided communication at the lower levels. We plan to address this issue by extending the CH3 API to include support for one-sided communication, particularly for networks that have native support for remote-memory access.

- On the Linux cluster, TCP socket communication is inherently two sided. As a result, it is difficult, if not impossible, to outperform MPI point-to-point communication because of the overhead (albeit small) added by the synchronization functions.

- Since we place no restrictions on the memory that a user can use for the memory window, the underlying communication even on shared-memory systems has two-sided semantics (copy into and out of shared memory). We plan to address this issue by optimizing the case where the user allocates the window with `MPI_Alloc_mem` (which can return a shared segment that a remote process can directly access), and, in some cases, by using operating-system hooks (such as in Linux) that enable a process to directly access the private memory of a remote process [5, 20]. (Note that for lock-unlock synchronization, an implementation is allowed to require the user to use window memory allocated with `MPI_Alloc_mem`.)

In a few cases in Tables 1 and 2, we see that the ratio is less than one, which means that one-sided communication is actually faster than point-to-point communication. We attribute this to the waiting time for the receive to be called in the rendezvous protocol used in point-to-point communication for large messages.

6 Conclusions and Future Work

This paper shows that an optimized implementation of the synchronization functions significantly improves the performance of MPI one-sided communication. Nonetheless, several opportunities exist for improving the performance further, and we plan to explore them. For example, in the case of lock-unlock synchronization for a single put or accumulate, we can improve the performance substantially by not having the origin process wait for an acknowledgment from

Table 1: Ratio of the time with one-sided communication to the time with point-to-point communication on the Linux cluster (the smaller the ratio, the better).

Size (bytes)	LAM fence	LAM pscw	MPICH2 fence	MPICH2 pscw
16	18.9	3.3	3.5	2.03
64	18.1	3.2	3.28	1.94
256	13.6	2.4	2.35	1.60
1K	5.1	1.52	1.59	1.39
16K	1.40	1.02	1.08	1.05
64K	1.03	0.95	0.85	0.78
256K	1.30	1.21	1.22	1.08

Table 2: Ratio of the time with one-sided communication to the time with point-to-point communication on the Sun SMP (the smaller the ratio, the better).

Size (bytes)	Sun MPI fence	Sun MPI pscw	Sun MPI lock	MPICH2 fence	MPICH2 pscw	MPICH2 lock
16	6.9	6.0	6.5	3.4	2.45	2.24
64	9.2	6.5	8.8	2.94	2.47	2.30
256	7.0	5.3	6.7	3.0	2.55	2.38
1K	2.0	1.55	1.93	2.43	2.06	1.92
16K	1.79	1.44	1.26	0.99	0.82	0.79
64K	1.48	2.16	1.54	1.13	1.06	0.77
256K	2.76	2.59	1.51	0.99	1.01	0.94

the target at the end of the unlock. This optimization, however, breaks the semantics of unlock, which state that when the unlock returns, the operation is complete at both origin and target. We plan to explore the possibility of allowing the user to pass an assert or info key to select weaker semantics that do not require the operation to be completed at the target when unlock returns. We plan to extend our implementation of one-sided communication (both synchronization and data transfer) to work efficiently on networks that have native support for remote-memory access, such as InfiniBand, Myrinet, and shared memory. We also plan to aggregate short messages and communicate them as a single message instead of separately, and cache derived datatypes at the target so that they need not be communicated each time.

Acknowledgments

We thank Chris Bischof for giving us access to the Sun SMP machines at the University of Aachen and Dieter an May for helping us in running our tests on those machines. We also thank Don Frederick for giving us access to the IBM SP at the San Diego Supercomputer Center.

References

[1] Noboru Asai, Thomas Kentemich, and Pierre Lagier. MPI-2 implementation on Fujitsu generic message passing kernel. In *Proceedings of SC99: High Performance Networking and Computing*, November 1999.

[2] David Ashton, William Gropp, Rajeev Thakur, and Brian Toonen. The CH3 design for a simple implementation of ADI-3 for MPICH-2 with a TCP-based implementation. Technical Report ANL/MCS-P1156-0504, Mathematics and Computer Science Division, Argonne National Laboratory, May 2004.

[3] Dan Bonachea. GASNet specification, v1.1. Technical Report CSD-02-1207, Dept. of Computer Science, UC Berkeley, October 2002.

[4] S. Booth and E. Mourão. Single sided MPI implementations for SUN MPI. In *Proceedings of SC2000: High Performance Networking and Computing*, November 2000.

[5] Phil Carns. Personal communication, 2004.

[6] Edgar Gabriel, Graham E. Fagg, and Jack J. Dongarra. Evaluating the performance of MPI-2 dynamic communicators and one-sided communication. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 10th European PVM/MPI Users' Group Meeting*, pages 88–97. Lecture Notes in Computer Science 2840, Springer, September 2003.

[7] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14):1947–1980, December 1998.

[8] Cray Research Inc. Cray T3E C and C++ optimization guide, 1994.

[9] Infiniband architecture. `http://www.infinibandta.org`.

[10] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, Darius Buntinas, Rajeev Thakur, and William Gropp. Efficient implementation of MPI-2 passive one-sided communication over InfiniBand clusters. In Dieter Kranzlmüller, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting*, pages 68–76. Lecture Notes in Computer Science 3241, Springer, September 2004.

[11] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, William Gropp, and Rajeev Thakur. High performance MPI-2 one-sided communication over InfiniBand. In *Proceedings of 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, April 2004.

[12] LAM/MPI Parallel Computing. `http://www.lam-mpi.org`.

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

[14] Elson Mourão and Stephen Booth. Single sided communications in multi-protocol MPI. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting*, pages 176–183. Lecture Notes in Computer Science 1908, Springer, September 2000.

[15] Fernando Elson Mourão and João Gabriel Silva. Implementing MPI's one-sided communications for WMPI. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users' Group Meeting*, pages 231–238. Lecture Notes in Computer Science 1697, Springer, September 1999.

[16] Myrinet software and documentation. `http://www.myri.com/scs`.

[17] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A non-uniform-memory-access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[18] Rajeev Thakur, William Gropp, and Brian Toonen. Minimizing synchronization overhead in the implementation of MPI one-sided communication. In Dieter Kranzlmüller, Peter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and*

Message Passing Interface, 11th European PVM/MPI Users' Group Meeting, pages 57–67. Lecture Notes in Computer Science 3241, Springer, September 2004.

[19] Jesper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Proceedings of SC2000: High Performance Networking and Computing*, November 2000.

[20] David Turner. Personal communication, 2004.

[21] Joachim Worringen, Andreas Gäer, and Frank Reker. Exploiting transparent remote memory access for non-contiguous and one-sided-communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC)*, April 2002.