An Evaluation of Implementation Options for MPI One-Sided Communication

William Gropp and Rajeev Thakur

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

Abstract. MPI defines one-sided communication operations—put, get, and accumulate—together with three different synchronization mechanisms that define the semantics associated with the initiation and completion of these operations. In this paper, we analyze the requirements imposed by the MPI Standard on any implementation of one-sided communication. We discuss options for implementing the synchronization mechanisms and analyze the cost associated with each. An MPI implementer can use this information to select the implementation method that is best suited (has the lowest cost) for a particular machine environment. We also report on experiments that we ran on a Linux cluster and a Sun SMP to determine the gap between the performance that could be achievable and what is actually achieved with MPI.

1 Introduction

Over the past decade, one-sided communication has emerged as a promising paradigm for high-performance communication on low-latency networks. The advantage of one-sided communication lies in its asynchronous nature: Unlike in point-to-point (or two-sided) communication where the sender and receiver explicitly call send and receive functions, in one-sided communication only the origin process calls the data-transfer function (put or get), and data transfer takes place without the target process explicitly calling any function to transfer the data. This model allows parallel programs to be less synchronizing and allows communication hardware to move data from one process to another with maximal efficiency. Nonetheless, some synchronization mechanism is needed in the programming model for the target process to indicate when its memory is ready for being read or written by a remote process and to specify when the data transfer is completed.

Because of the growing popularity of one-sided communication, the MPI Forum defined a specification for one-sided communication in MPI as part of the MPI-2 Standard [11]. MPI defines three data-transfer functions for one-sided communication: put (remote write), get (remote read), and accumulate (remote update). These data-transfer functions must be used together with one of three synchronization mechanisms—fence, post-start-complete-wait, and lock-unlock—as shown in Figure 1. The memory region that a process exposes to one-sided communication is called a *window*, and a collection of processes create a *window object* that is used as a handle in all data transfer and synchronization

```
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

Fig. 1. The three synchronization mechanisms for one-sided communication in MPI. The numerical arguments indicate the target rank.

calls. Puts, gets, and accumulates are nonblocking operations, and an implementation is allowed to reorder them within a synchronization epoch. They are guaranteed to be completed (both locally and remotely) only when the synchronization epoch has ended.

The synchronization mechanisms, although necessary in a programming model, add overhead to an implementation of one-sided communication. In this paper, we analyze the requirements imposed by the MPI Standard on any implementation of one-sided communication and associated synchronization. We discuss different options for implementing the synchronization mechanisms and analyze the cost associated with each of them. An MPI implementer can use this information to choose the implementation method best suited for a particular machine environment. We also report on experiments that we ran on a Linux cluster and a Sun SMP to determine the gap between the performance that could be achievable and what is actually achieved with MPI.

2 Related Work

One-sided communication is supported by many low-level communication libraries, such as GASNET [2], ARMCI [14], and Portals [4], as well as by user-level programming models such as BSP [7], Global Arrays [15], and SHMEM [8]. Moreover, many MPI implementations, including all vendor MPIs, support one-sided communication, with varying levels of optimization [1, 3, 9, 10, 12, 13, 17, 19, 20]. Nonetheless, Gabriel et al. [5] found that, because of the synchronization overhead in one-sided communication, regular point-to-point communication performs better than one-sided communication in five MPI implementations: NEC, Hitachi, IBM, Sun, and LAM. The only exceptions were NEC and Sun MPI when window memory allocated with the special function `MPI_Alloc_mem` is used. Clearly, it is necessary to analyze the costs associated with the synchronization mechanisms and optimize the implementations.

3 Fence Synchronization

Figure 1a illustrates the fence method of synchronization. `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 call to fence completes the one-sided operations issued by this process as well as the operations targeted at this process by other processes. An implementation of fence synchronization must support the following semantics: A one-sided operation cannot access a process's window until that process has called fence, and the second fence on a process cannot return until all processes needing to access that process's window have completed doing so.

3.1 Implementing Fence

In general, an implementation has two options for implementing fence: *immediate* and *deferred*.

Immediate Method. This method implements the synchronization and communication operations as they are issued. A simple implementation of this option is to perform a barrier in the first fence; perform the puts, gets, and accumulates as they are called; and perform another barrier at the end of the second fence after all the one-sided operations have completed. The first barrier ensures that all processes know that all other processes have reached the first fence and that it is now safe to access their windows. The second barrier ensures that a process does not return from the second fence until all other processes have finished accessing its window.

On a distributed-memory environment without hardware support for barriers, a barrier can be implemented by using the dissemination algorithm [6] with 0-byte messages. If p is the number of processes and α is the latency (or startup time) per message, this algorithm costs $(\lg p)\alpha$. The immediate method requires two barriers, which cost $2(\lg p)\alpha$. This method is expensive in environments where the latency is high, such as on clusters and networks running TCP. It is appropriate for environments where barriers can be fast, such as shared-memory systems or machines with hardware support for barriers, such as the Cray T3E and IBM BG/L.

Deferred Method. In this method, the first fence does nothing and simply returns. The ensuing puts, gets, and accumulates are simply queued up locally. All the work is done in the second fence, where each process first 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 otherwise. All processes then do a reduce-scatter sum operation on this array (as in `MPI_Reduce_scatter`). As a result, each process knows how many processes will be performing one-sided operations on its window, and this number is stored in a counter in the window object. Each process then performs the data transfer for its one-sided operations and ensures that the counter at the target is decremented when all the one-sided operations from this process to that target have been completed. As a result, a process can return from the second fence when the one-sided operations issued by that process have completed locally and when the counter in its window object reaches 0, indicating that all other processes have finished accessing its window.

This method thus eliminates the need for a barrier in the first fence and replaces 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 (asynchronously).

On a distributed-memory system, a reduce-scatter operation on an array of p short integers (2 bytes) costs $(\lg p)\alpha + 2(p-1)\beta$ [18], where β is the transfer time per byte between two processes. We ignore the computation time for the reduction because the array is small. Because of the lower latency term, this method is preferred over the immediate method on systems where the latency is relatively high, such as on clusters.

3.2 Performance

To determine how MPI implementations perform for fence synchronization, we measured the cost of two `MPI_Barrier`s, one `MPI_Reduce_scatter`, and two `MPI_Win_fence`s on a Myrinet-connected Linux cluster at Argonne and a Sun SMP at the University of Aachen in Germany. On the Linux cluster, we used the latest CVS version of MPICH2 (version 1.0.1 with some enhancements) with the GASNET channel on top of GM. On the Sun SMP, we used Sun MPI. We performed the operations several times in a loop, calculated the average time for one iteration, and then the maximum time taken by all processes. We used `MPI_Alloc_mem` to allocate window memory and passed the following asserts to the two fences:

```
MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
MPI_Win_fence(MPI_MODE_NOSTORE | MPI_MODE_NOPUT |
              MPI_MODE_NOSUCCEED, win);
```

Figure 2 shows the results. On the Linux cluster, the cost of two barriers is far more than the cost of a single reduce-scatter. Therefore, the deferred method is the preferred option, which is what MPICH2 uses in this case. We see that the cost of two fences is almost the same as that of a reduce-scatter. On the Sun SMP, Sun MPI has a very fast implementation of barrier, and therefore the immediate method is the preferred implementation strategy for fence. From the

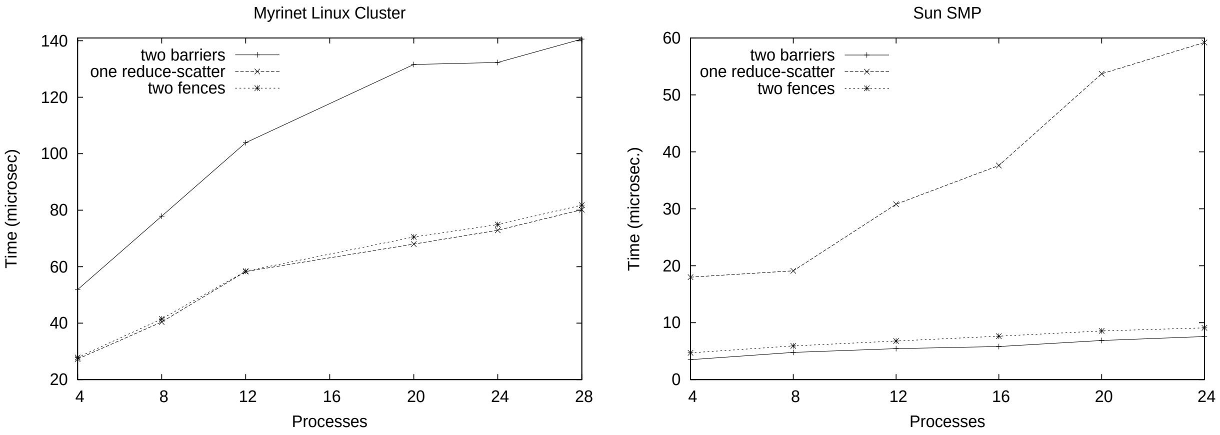

Fig. 2. Time taken for two barriers, one reduce-scatter, and two fences on a Myrinet-connected Linux cluster using MPICH2 (left) and on a Sun SMP using Sun MPI (right).

graph, it appears that Sun MPI does use the immediate method, because the time for two fences is only slightly higher than the time for two barriers.

4 Post-Start-Complete-Wait Synchronization

The drawback of fence synchronization is that it is collective over the entire communicator associated with the window object, which results in unnecessary synchronization overhead when only small subsets of processes are actually communicating with each other. To avoid this drawback, MPI defines a second synchronization mechanism in which only subsets of processes need to synchronize, as shown in Figure 1b. 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.

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`.

4.1 Implementing Post-Start-Complete-Wait

An implementation again has two options to implement post-start-complete-wait: immediate and deferred.

Immediate Method. In this method, `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 are performed as they are called. 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. On the target, MPI_Win_wait blocks until it receives the done message from each origin process. Assuming that the size of the origin and target groups is g, the overhead of this method is $2g\alpha$. This method is appropriate in environments with low latency and native support for one-sided communication, such as shared memory, so that the data transfer can be initiated as soon as it is called.

Deferred Method. This method defers data transfer until the second synchronization call. 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 its window object to the size of this group. On the origin side, MPI_Win_start does nothing and simply returns. All one-sided operations are simply queued up locally. In MPI_Win_complete, the origin process first waits to receive the zero-byte messages from the processes in the target group. It then performs all the one-sided operations and ensures that the window counter at the target gets decremented when all the one-sided operations from this process to that target have been completed. MPI_Win_complete returns when all its operations have locally completed. On the target, MPI_Win_wait simply blocks and invokes the progress engine until its window counter reaches zero, indicating that all origin processes have finished accessing its window.

Thus the only synchronization in this method 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). If the size of the origin and target groups is g, the overhead of this method is $g\alpha$. Therefore, the deferred method is faster in environments where latency is high.

5 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 1c. After issuing the one-sided operations, it calls MPI_Win_unlock. The target does not make any synchronization call. When MPI_Win_unlock returns, the one-sided operations are guaranteed to be completed at the origin *and* the target. 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. 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 [11].

5.1 Implementing Lock-Unlock

An implementation again has two options to implement lock-unlock.

Immediate Method. In this method, MPI_Win_lock sends a lock-request packet to the target and waits for a lock-granted reply. Puts, gets, and accumulates are performed as they are called. MPI_Win_unlock waits until all one-sided operations initiated by that process have completed locally and then sends an unlock request to the target. It also waits to receive an acknowledgment from the target that all the one-sided operations issued from this process have completed at the target, as required by the semantics of MPI_Win_unlock. The cost for acquiring the lock is 2α, and the cost for releasing the lock is also 2α. Therefore, the total cost of this method is 4α, assuming no lock contention.

Deferred Method. In this method, MPI_Win_lock does nothing and simply returns. All one-sided operations are simply queued up locally. In MPI_Win_unlock, the origin sends a lock-request packet to the target and waits for a lock-granted reply. It then performs the one-sided operations. When these operations have completed locally, it sends an unlock request to the target and, in the general case, waits for a reply from the target indicating that the operations have completed at the target.

The deferred method also costs 4α in the general case. This method, however, permits several optimizations that the immediate method does not. One optimization is that if any of the one-sided operations is a get, it can be reordered and performed last. Since the origin must wait to receive data in the get, when the get completes, it implies that the one-sided operations have also completed at the target (assuming ordered completion). In this case, an additional acknowledgment is not needed from the target, thereby reducing the cost to 3α. Another optimization in the case of a single put, get, or accumulate is that the origin can perform it as an atomic lock-(put/get/accumulate)-unlock request without having to wait for a lock-granted reply. If the operation is a get, even the additional completion acknowledgment from the target is not needed. These optimizations reduce the cost of lock-unlock to α and 0, respectively, because the lock request becomes part of the data transfer.

6 Analysis for Shared-Memory Environments

Shared-memory environments offer unique opportunities for optimizing MPI one-sided communication because of their support for atomic operations for fast synchronization and the ability to directly copy data to/from the user's buffer on the target if the window memory was allocated with the function MPI_Alloc_mem. We analyze in further detail the implementation of one-sided communication on shared-memory environments. For lack of space, we focus only on lock-unlock synchronization. We consider in detail the steps that an MPI implementation is required to take to enforce the semantics of lock-unlock assuming the immediate method of implementation. We then compare these steps to an OpenMP [16] program that performs the same operation by using shared memory directly, which forms a baseline for the performance that can be achieved.

Consider this simple example that puts n longs into the memory window on the process specified by rank:

```
    MPI_Win_lock(MPI_MODE_EXCLUSIVE, rank, 0, win);
    MPI_Put(buf, n, MPI_LONG, rank, 0, n, MPI_LONG, win);
    MPI_Win_unlock(rank, win);
```

We assume that the window memory was allocated with MPI_Alloc_mem and is directly accessible by a remote process. If we ignore error checking of function parameters, an MPI implementation need perform only the following steps for each of the above functions.

MPI_Win_lock

1. Make a routine call with four arguments
2. Convert win into an address (if not already an address)
3. Look up the address of the lock at the target (indexed access into win)
4. Check shared or exclusive access
5. Remote update for the lock

MPI_Win_put

1. Make a routine call with eight arguments
2. Convert win into an address (if not already an address)
3. Check that the remote memory is directly accessible
4. Get the base address of the remote memory (indexed access into win)
5. Get the displacement unit (indexed access into win)
6. Determine whether origin data is contiguous and get length (access datatype and multiply count by datatype size)
7. Determine whether target data is contiguous
8. Perform the copy of local to remote memory

MPI_Win_unlock

1. Make a routine call with two arguments
2. Convert win into an address (if not already an address)
3. Look up the address of the lock at the target (indexed access into win)
4. Remote update for the unlock

While the number of steps may seem large, they in fact involve relatively few instructions. However, the access to remote memory, either for the lock accesses or for the memory copy at the end of the MPI_Put step, may require hundreds of processor cycles. To determine the cost of performing the above steps, we wrote a shared-memory program using OpenMP [16] in which one thread performs the equivalent of lock, put, and unlock on another thread. We wrote four versions of this program:

1. A single routine OpenMP program where the lock, put, and unlock are all performed in the main routine by simply setting and clearing a flag for the lock and unlock steps and using memcpy to move the data.
2. The lock, put, and unlock operations are performed in separate routines, thus adding function-call overhead.
3. An MPI_Win-like structure is added so that the addresses of the lock and the memory at the target have to be obtained by indexing into the structure.
4. The routines use the same arguments as the corresponding MPI functions. This version adds more arguments to the routines and requires an extra lookup in the win structure for the displacement unit at the target.

We ran these programs on a Sun SMP at the University of Aachen with the Sun OpenMP compiler. This machine is a Sun Fire E6900 with a 1.2 GHz Ultra Sparc IV. We also ran the MPI version of the program, in which the window memory was allocated with `MPI_Alloc_mem`. We used two MPI implementations: Sun MPI and the latest CVS version of MPICH2 (version 1.0.1 with some enhancements) with the sshm (scalable shared memory) channel. For windows allocated with `MPI_Alloc_mem`, this version of MPICH2 uses the immediate method to implement all three synchronization methods in the sshm channel.

Table 1 shows the time taken to move 8, 256, 1024, and 64K `longs` (4 bytes) using these six programs. The results show that the MPI implementations of the one-sided operations are not substantially more expensive than the equivalent operations performed with OpenMP, even if compared with the first version of the Open MP program where many of the MPI features are not present. Nonetheless, a difference of a factor of two to three exists between the OpenMP and MPI results. By the final version of this paper, we plan to determine the source of this difference. Preliminary studies indicate that the cost of `MPI_Win_lock` followed by `MPI_Win_unlock` is itself roughly twice as large as the equivalent steps in the OpenMP version. This may be due to the difference in the handling of thread and process locks in the operating system, and we plan to investigate this issue further.

Table 1. Time in seconds to perform a lock-put-unlock operation on a Sun SMP. n is the number of `longs` (4 bytes) moved. The column "OpenMP 1" is a simple OpenMP implementation of this operation. The other three OpenMP programs additional features. The column "OpenMP 4" is an OpenMP program that mimics the steps an MPI implementation must implement. The last two columns show the times for two MPI implementations on this platform, Sun MPI and MPICH2.

n	OpenMP 1 (simple)	OpenMP 2 (routines)	OpenMP 3 (win struct)	OpenMP 4 (all MPI args)	Sun MPI	MPICH2
8	4.5e-7	4.8e-7	4.9e-7	5.4e-7	1.1e-6	1.3e-6
256	1.1e-6	1.1e-6	1.1e-6	1.2e-6	1.7e-6	1.9e-6
1024	4.7e-6	4.9e-6	5.0e-6	5.1e-6	5.2e-6	6.6e-6
64K	2.5e-4	2.6e-4	2.6e-4	2.7e-4	4.3e-4	5.4e-4

7 Conclusions and Future Work

MPI one-sided communication has the potential to deliver high performance to applications. However, MPI implementations need to implement it efficiently, with particular emphasis on minimizing the overhead added by the synchronization functions. In this paper, we have analyzed the minimum requirements an implementation must meet to honor the semantics specified by the MPI Standard. We have discussed and analyzed several implementation options and recommended which option to use in which environments. Our analysis of the performance of lock-put-unlock on the Sun SMP demonstrates that MPI implementations are not too far off from delivering what can be delivered by using direct shared memory, although there is room for improvement. We plan to investigate in further detail where the additional gap lies and how much of it can be reduced with clever implementation strategies.

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 W-31-109-ENG-38. We thank Chris Bischof for giving us access to the Sun SMP machines at the University of Aachen.

References

1. Noboru Asai, Thomas Kentemich, and Pierre Lagier. MPI-2 implementation on Fujitsu generic message passing kernel. In *Proc. of SC99: High Performance Networking and Computing*, November 1999.
2. Dan Bonachea. GASNet specification, v1.1. Technical Report CSD-02-1207, Dept. of Computer Science, UC Berkeley, October 2002.
3. S. Booth and E. Mourão. Single sided MPI implementations for SUN MPI. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.
4. Ron Brightwell, Rolf Riesen, Bill Lawry, and A. B. Maccabe. Portals 3.0: Protocol building blocks for low overhead communication. In *Proc. of the 2002 Workshop on Communication Architecture for Clusters*, 2002.
5. Edgar Gabriel, Graham E. Fagg, and Jack J. Dongarra. Evaluating dynamic communicators and one-sided operations for current MPI libraries. *Int'l Journal of High-Performance Computing Applications*, 19(1):67–80, Spring 2005.
6. Debra Hensgen, Raphael Finkel, and Udi Manbet. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.
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. 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 *Proc. of the 11th European PVM/MPI Users' Group Meeting*, pages 68–76, September 2004.
10. 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 *Proc. of the 4th Int'l Symp. on Cluster Computing and the Grid (CCGrid 2004)*, April 2004.
11. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997. `http://www.mpi-forum.org/docs/docs.html`.
12. Elson Mourão and Stephen Booth. Single sided communications in multi-protocol MPI. In *Proc. of the 7th European PVM/MPI Users' Group Meeting*, pages 176–183, September 2000.
13. Fernando Elson Mourão and João Gabriel Silva. Implementing MPI's one-sided communications for WMPI. In *Proc. of the 6th European PVM/MPI Users' Group Meeting*, pages 231–238, September 1999.
14. Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Proc. of the 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) at IPDPS/SPDP '99*, April 1999.
15. 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.
16. OpenMP. `http://www.openmp.org`.
17. Rajeev Thakur, William Gropp, and Brian Toonen. Minimizing synchronization overhead in the implementation of MPI one-sided communication. In *Proc. of the 11th European PVM/MPI Users' Group Meeting*, pages 57–67, September 2004.
18. Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in MPICH. *Int'l Journal of High-Performance Computing Applications*, 19(1):49–66, Spring 2005.
19. Jesper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.
20. Joachim Worringen, Andreas Gäer, and Frank Reker. Exploiting transparent remote memory access for non-contiguous and one-sided-communication. In *Proc. of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.