Analysis of Implementation Options for MPI-2 One-Sided

Brian W. Barrett[1], Galen M. Shipman[1], Andrew Lumsdaine[2],

[1] Los Alamos National Laboratory [⋆], Los Alamos, NM 87545, USA,
{bbarrett, gshipman}@lanl.gov
[2] Indiana University [⋆⋆], Bloomington, IN 47405, USA,
lums@osl.iu.edu

Abstract. The Message Passing Interface provides an interface for one-sided communication as part of the MPI-2 standard. The semantics specified by MPI-2 allow for a number of different implementation avenues, each with different performance characteristics. Within the context of Open MPI, a freely available high performance MPI implementation, we analyze a number of implementation possibilities, including layering over MPI-1 send/receive and true remote memory access.

1 Introduction

The Message Passing Interface [1,2,3,4] (MPI) has been adopted by the high performance computing community as the communication library of choice for distributed memory systems. The original MPI specification provides for point-to-point and collective communication, as well as environment management functionality. The MPI-2 specification added dynamic process creation, parallel I/O, and one-sided communication.

The MPI-2 one-sided specification allows for implementation over send/receive or remote memory access (RMA) networks. Although this design feature has been the source of criticism [5], it also ensures maximum portability, a goal of MPI. The MPI-2 one-sided interface utilizes the concept of exposure and access epochs to define when communication can be initiated and when it must be completed. Explicit synchronization calls are used to initiate both epochs, a feature which presents a number of implementation options, even when networks support true RMA operations. This paper examines implementation options for the one-sided interface within the context of Open MPI.

[⋆] Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. LA-UR-07-3197.

[⋆⋆] This work was supported by a grant from the Lilly Endowment and National Science Foundation grants EIA-0202048 and ANI-0330620.

2 Related Work

A number of MPI implementations provide support for the one-sided interface. LAM/MPI [6] provides an implementation layered over point-to-point, although it does not support passive synchronization and performance generally does not compare well with other MPI implementations. Sun MPI [7] provides a high performance implementation, although it requires all processes be on the same machine and the use of MPI_ALLOC_MEM for optimal performance. The NEC SX-5 MPI implementation includes an optimized implementation utilizing the global shared memory available on the platform [8]. The SCI-MPICH implementation provides one-sided support using hardware reads and writes [9].

MPICH2 [10] includes a one-sided implementation implemented over point-to-point and collective communication. Lock/unlock is supported, although the passive side must enter the library to make progress. The synchronization primitives in MPICH2 are significantly optimized compared to previous MPI implementations [11]. MVAPICH2 [12] extends the MPICH2 one-sided implementation to utilize InfiniBand's RMA support. MPI_PUT and MPI_GET communication calls translate into InfiniBand put and get operations for contiguous datatypes. MVAPICH2 has also examined using native InfiniBand for Lock/Unlock synchronization [13].

3 Open MPI Architecture

Open MPI [14] is a complete, open-source MPI implementation. The project is developed as a collaboration between a number of academic, government, and commercial institutions, including Indiana University, University of Tennessee, Knoxville, University of Houston, Los Alamos National Laboratory, Cisco Systems, and Sun Microsystems. Open MPI is designed to be scalable, fault tolerant, and high performance, while at the same time being portable to a variety of networks and operating systems. Open MPI utilizes a low-overhead component architecture—the Modular Component Architecture (MCA)—to provide abstractions for portability and adapting to differing application demands. In addition to providing a mechanism for portability, the MCA allows developers to experiment with different implementation ideas, while minimizing development overhead.

MPI communication is layered on a number of component frameworks, as shown in Fig. 1. The PML and OSC frameworks provide MPI send/receive and one-sided semantics, respectively. The BML, or BTL Management Layer, allows the use of multiple networks between two processes and the use of multiple upper-layer protocols on a given network, by maintainin available routes to every peer and handling scheduling across available routes. The BTL framework provides communication between two endpoints; an endpoint is usually a communication device connecting two processes, such as an Ethernet address or an InfiniBand port. The current BML/BTL implementation allows multiple upper-layer protocols to simultaneously utilize multiple communication paths. The design and implementation of the communication layer is described in detail in [15,16].

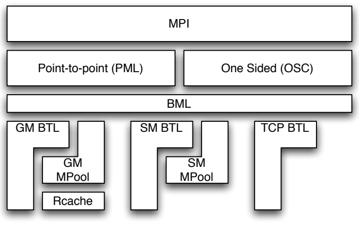


Fig. 1. Component structure for communication in Open MPI.

BTL components provide two communication modes: an active-message style send/receive protocol and a remote memory access (RMA) put/get protocol. All sends are non-blocking, with a callback on local send completion, generally from a BTL-provided buffer. Receives are all into BTL-provided buffers, with a callback on message arrival. RMA operations provide callbacks on completion on the origin process and no completion callbacks on the target process (as all networks do not support remote completion events for RMA operations). All buffers used on both the origin and target must be "prepared" for use by calls to the BTL by higher-level components.

4 Open MPI One-sided Implementation

Open MPI provides two implementations of the one-sided (OSC) framework: pt2pt and rdma. The pt2pt component is implemented entirely over the point-to-point and collective MPI functions. The original one-sided implementation in Open MPI, it is now primarily used when a network library does not expose RMA capabilities, such as Myrinet MX [17]. The rdma component is implemented directly over the BML/BTL interface and supports a variety of protocols, including active-message send/receive and true RMA. More detail on the implemnetation is provided in Section 4.2. Both components share the same synchronization implementation, although the rdma component starts communication before the synchronization call to end an epoch, while the pt2pt component does not.

4.1 Synchronization

Synchronization for both components is similar to the design used by MPICH2 [11]. Control messages are sent either over the point-to-point engine (pt2pt) or the BTL (rdma). A brief overview of the synchronization implementation follows:

Fence MPI_WIN_FENCE is implemented with a reduce-scatter to share the number of incoming communication operations, then each process waits until the specified number of operations has completed. Communication may be started at any time during the exposure/access epoch.

General Active Target A call to MPI_POST results in a post control message sent to every involved process. Communication may be started as soon as a post message is received from all involved processes. During MPI_COMPLETE, all RMA operations are completed, then a control message with the number of incoming requests is sent to all peer processes. MPI_WAIT blocks until all peers send a complete message and incoming operations are completed.

Passive Lock/Unlock synchronization does not wait for a lock to be acquired before returning from MPI_WIN_LOCK, but may start all communication as soon as a lock acknowledgment is received. During MPI_WIN_UNLOCK, a control message with number of incoming messages is sent to the peer. The peer waits for all incoming messages before releasing the lock and potentially giving it to another peer. Like other single threaded MPI implementations, our implementation currently requires the target process enter the MPI library for progress to be made.

4.2 Communication

Three communication protocols are implemented for the rdma one-sided component. For networks that support RMA operations, all three protocols are available at run-time, and the selection of protocol is made per-message.

send/recv All communication is done using the send/receive interface of the BTL, with data copied at both sides for short messages. Long messages are transferred using the protocols provided by the PML (which may include RMA operations). Messages are queued until the end of the synchronization phase.

buffered All communication is done using the send/receive interface of the BTL, with data copied at both sides for short messages. Long messages are transferred using the transfer protocols provided by the PML. Short messages are coalesced into the BTL's maximum eager send size. Messages are started as soon as the synchronization phase allows.

RMA All communication for contiguous data is done using the RMA interface of the BTL. All other data is transferred using the *buffered* protocol. MPI_ACCUMULATE also falls back to the *buffered* protocol.

Due to the lack of remote completion notification for RMA operations, care must be taken to ensure that an epoch is not completed before all data transfers have been completed. Because ordering semantics of RMA operations (especially compared to send/receive operations) tends to vary widely between network interfaces, the only ordering assumed by the rdma component is that a message sent after local completion of an RMA operation will result in remote completion of the send after the full RMA message has arrived. Therefore, any completion messages sent during synchronization may only be sent after all RMA operations to a given peer have completed. This is a limitation in performance for some networks, but adds to the overall portability of the system.

5 Performance Evaluation

Latency and bandwidth micro-benchmark results are provided using the Ohio State benchmark suite. Unlike the point-to-point interface, the MPI community has not developed a set of standard "real world" benchmarks for one-sided communication. Following previous work [11], a nearest-neighbor ghost cell update benchmark is utilized — the fence version is shown in Fig. 5. The test was also extended to call MPI_PUT once per integer in the buffer, rather than once per buffer.

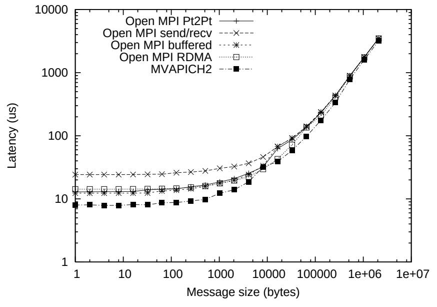
```
for (i = 0 ; i < ntimes ; i++) {
  MPI_Win_fence(MPI_MODE_NOPRECEEDE, win);
  for (j = 0 ; j < num_nbrs ; j++) {
    MPI_Put(send_buf + j * bufsize, bufsize, MPI_DOUBLE, nbrs[j],
            j, bufsize, MPI_DOUBLE, win);
  }
  MPI_Win_fence(0, win);
}
```

Fig. 2. Ghost cell update using MPI_FENCE

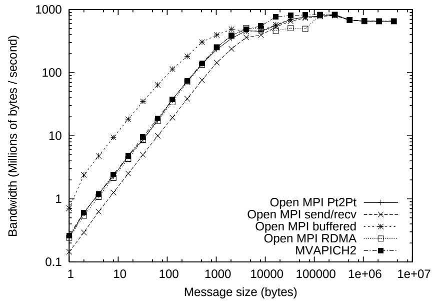
All tests were run on the Indiana University Department of Computer Science Odin cluster, a 128 node cluster of dual-core dual-socket 2.0 GHz Opteron machines, each with 4 GB of memory. Each node contains a single Mellanox InfiniHost PCI-X SDR HCA, connected to a 148 port switch. MVAPICH2 0.9.8 results are provided as a baseline. No configuration or run-time performance options were specified for MVAPICH2. The *mpi_leave_pinned* option, which tells Open MPI to leave memory registered with the network until the buffer is freed by the user rather than when communication completes, was specified to Open MPI (this functionality is the default in MVAPICH). Results are provided for the pt2pt component and all three protocols of the rdma component.

Latency/Bandwidth Fig. 3 presents the latency and bandwidth of MPI_PUT using the Ohio State benchmarks [18]. The *buffered* protocol presents the best latency for Open MPI. Although the message coalescing of the *buffered* protocol does not improve performance of the latency test, due to only one message pending during an epoch, the protocol outperforms the *send/recv* protocol due to starting messages eagerly, as soon as all post messages are received. The *buffered* protocol provides lower latency than the *rdma* protocol for short messages because of the requirement for portable completion semantics, described in the previous section. No completion ordering is required for the *buffered* protocol, so MPI_WIN_COMPLETE does not wait for local completion of the data transfer before sending the completion count message. On the other hand, the *rdma* protocol must wait for local completion of the event before sending the completion count control message, otherwise the control message could overtake the RDMA transfer, resulting in erroneous results.

The bandwidth benchmark shows the advantage of the *buffered* protocol, as the benchmark starts many messages in each synchronization phase. The *buffered* protocol is therefore able to outperform both the *rdma* protocol and MVAPICH. Again, the *send/recv* protocol suffers compared to the other protocols, due to the extra copy overhead compared to *rdma*, the extra transport headers compared to both *rdma* and *buffered*, and the delay in starting data transfer until the end of the synchronization phase. For large messages, where all protocols are utilizing RMA operations, realized bandwidth is similar for all implementations.

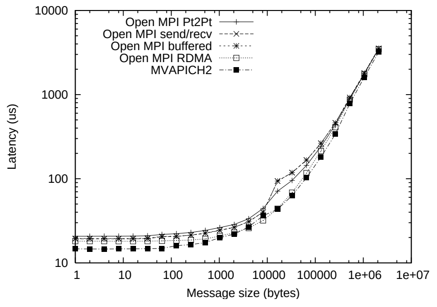


(a) Latency

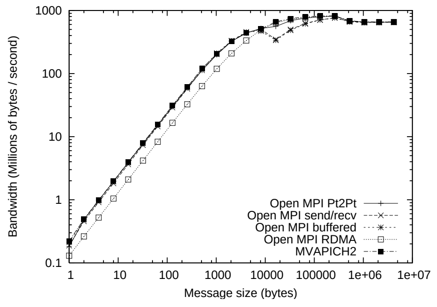


(b) Bandwidth

Fig. 3. Latency and Bandwidth of MPI_PUT calls between two peers using generalized active synchronization.



(a) Latency



(b) Bandwidth

Fig. 4. Latency and Bandwidth of MPI_GET calls between two peers using generalized active synchronization.

The latency and bandwidth of MPI_GET are shown in Fig. 4. The *rdma* protocol has lower latency than the send/receive based protocols, as the target process does not have to process requests at the MPI layer. The present *buffered* protocol does not coalesce reply messages from the target to the origin, so there is little advantage to using the *buffered* protocol over the *send/recv* protocol. For the majority of the bandwidth curve, all implementations other than the *rdma* protocol provide the same bandwidth. The *rdma* protocol clearly suffers from a performance issue that the MVAPICH2 implementation does not. For short messages, we believe the performance lag is due to receiving the data directly into the user buffer, which requires registration cache look-ups, rather than copying through a pre-registered “bounce” buffer. The use of a bounce buffer for MPI_PUT but not MPI_GET is an artifact of the BTL interface, which we are currently addressing.

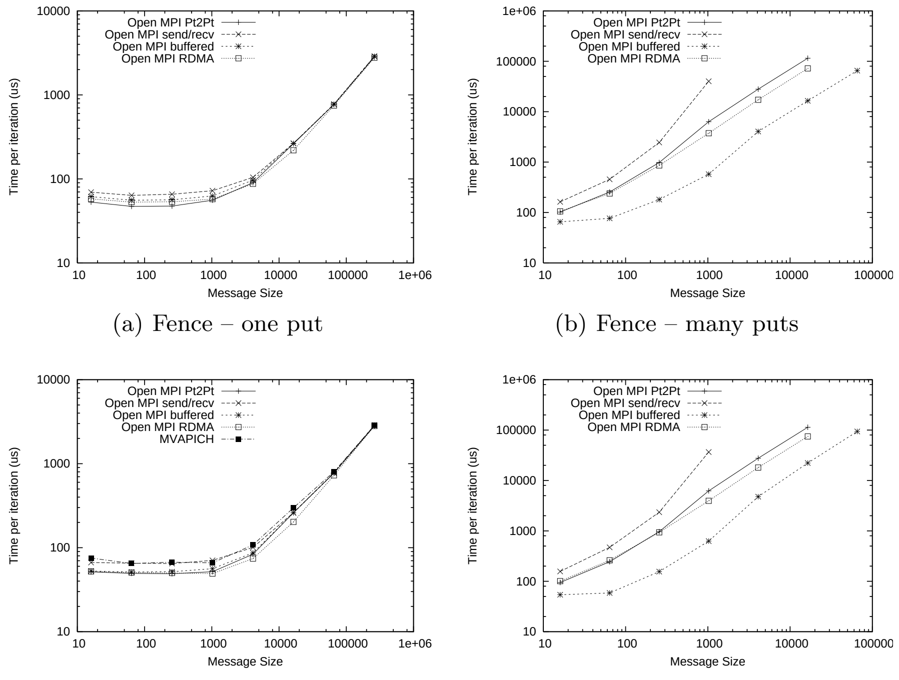
Ghost Cell Updates Fig. 5 shows the cost of performing an iteration of a ghost cell update sequence. The tests were run across 32 nodes, one process per node. For both fence and generalized active synchronization, the ghost cell update with large buffers shows relative performance similar to the put latency shown previously. This is not unexpected, as the benchmarks are similar with the exception that the ghost cell updates benchmark sends to a small number of peers rather than to just one peer. Fence results are not shown for MVAPICH2 because the tests ran significantly slower than expected and we suspect that the result is a side effect of the testing environment.

When multiple puts are intiated to each peer, the benchmark results show the disadvantage of the *send/recv* and *rdma* protocol compared to the *buffered* protocol. The number of messages injected into the MPI layer grows as the message buffer grows. With larger buffer sizes, the cost of creating requests, buffers, and the poor message injection rates of InfiniBand becomes a limiting factor. When using InfiniBand, the *buffered* protocol is able to reduce the number of messages injected into the network by over two orders of magnitude.

6 Summary

As we have shown, there are a number of implementation options for the MPI one-sided interface. While the general consensus in the MPI community has been to exploit the RMA interface provided by modern high performance networks, our results appear to indicate that such a decision is not necessarily clear-cut. The message coalescing opportunities available when using send/receive semantics provides much higher realized network bandwidth than when using RMA. The completion semantics imposed by a portable RMA abstraction also requires ordering that can cause higher latencies for RMA operations than for send/receive semantics.

Using RMA operations has one significant advantage over send/receive – the target side of the operation does not need to be involved in the message transfer, so the theoretical availability of computation/communication overlap is



(a) Fence – one put (b) Fence – many puts

(c) Generalized – one put (d) Generalized – many puts

Fig. 5. Ghost cell iteration time at 32 nodes for varying buffer size, using fence or generalized active synchronization.

improved. In our tests, we were unable to see this in practice, likely due less to any shortcomings of RMA and more due to the two-sided nature of the MPI-2 one-sided interface. Further, we expected the computation/communication overlap advantage to become less significant as Open MPI develops a stronger progress thread model, allowing message unpacking as messages arrive, regardless of when the application enters the MPI library.

References

1. Geist, A., Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Saphir, W., Skjellum, T., Snir, M.: MPI-2: Extending the Message-Passing Interface. In: Euro-Par '96 Parallel Processing, Springer Verlag (1996) 128–135
2. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI: The Complete Reference: Volume 2, the MPI-2 Extensions. MIT Press (1998)
3. Message Passing Interface Forum: MPI: A Message Passing Interface. In: Proc. of Supercomputing '93, IEEE Computer Society Press (November 1993) 878–883
4. Snir, M., Otto, S.W., Huss-Lederman, S., Walker, D.W., Dongarra, J.: MPI: The Complete Reference. MIT Press, Cambridge, MA (1996)

5. Bonachea, D., Duell, J.: Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. Int. J. High Performance Computing and Networking **1**(1/2/3) (2004) 91–99
6. Burns, G., Daoud, R., Vaigl, J.: LAM: An Open Cluster Environment for MPI. In: Proceedings of Supercomputing Symposium. (1994) 379–386
7. Booth, S., Mourao, F.E.: Single Sided Implementations for SUN MPI. In: Supercomputing. (2000)
8. Trff, J.L., Ritzdorf, H., Hempel, R.: The implementation of mpi-2 one-sided communication for the nec sx-5. In: Supercomputing 2000, IEEE/ACM (2000)
9. Worringen, J., Ger, A., Reker, F.: Exploiting transparent remote memory access for non-contiguous and one-sided-communication. In: Proceedings of ACM/IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002), Workshop for Communication Architecture in Clusters (CAC 02), Fort Lauderdale, USA (April 2002)
10. Argonne National Lab.: MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2/
11. Thakur, R., Gropp, W., Toonen, B.: Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication. Int. J. High Perform. Comput. Appl. **19**(2) (2005) 119–128
12. Huang, W., Santhanaraman, G., Jin, H.W., Gao, Q., Panda, D.K.: Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand. In: Int'l Sympsoium on Cluster Computing and the Grid (CCGrid), Singapore (May 2006)
13. Jiang, W., Liu, J., Jin, H.W., Panda, D.K., Buntinas, D., Thakur, R., Gropp, W.: Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (September 2004)
14. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (September 2004) 97–104
15. Shipman, G.M., Woodall, T.S., Graham, R.L., Maccabe, A.B., Bridges, P.G.: InfiniBand Scalability in Open MPI. In: IEEE International Parallel And Distributed Processing Symposium. (2006 (to appear))
16. Woodall, T., et al.: Open MPI's TEG point-to-point communications methodology : Comparison to existing implementations. In: Proceedings, 11th European PVM/MPI Users' Group Meeting. (2004)
17. Myricom, Inc: Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet (2006)
18. Network-Based Computing Laboratory, Ohio State University: Ohio State Benchmark Suite. http://mvapich.cse.ohio-state.edu/benchmarks/