Revealing the Performance of MPI RMA Implementations

William D. Gropp and Rajeev Thakur

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

Abstract. The MPI remote-memory access (RMA) operations provide a different programming model from the regular MPI-1 point-to-point operations. This model is particularly appropriate for cases where there are multiple communication events for each synchronization and where the target memory locations are known by the source processes. In this paper, we describe a benchmark designed to illustrate the performance of RMA with multiple RMA operations for each synchronization, as compared with point-to-point communication. We measured the performance of this benchmark on several platforms (SGI Altix, Sun Fire, IBM SMP, Linux cluster) and MPI implementations (SGI, Sun, IBM, MPICH2, Open MPI). We also investigated the effectiveness of the various optimization options specified by the MPI standard. Our results show that MPI RMA can provide substantially higher performance than point-to-point communication on some platforms, such as SGI Altix and Sun Fire. The results also show that many opportunities still exist for performance improvements in the implementation of MPI RMA.

1 Introduction

MPI-2 added remote-memory access (RMA) operations to the MPI standard. These one-sided operations offer the promise of improved performance for applications, yet users are uncertain whether these operations offer any advantage in most implementations.

A key feature of the one-sided operations is that data transfer and synchronization are separated. This allows multiple transfers to use a single synchronization operation, thus reducing the total overhead. RMA differs from the two-sided or point-to-point model, where each message combines both the transfer and the synchronization. Because of this feature, a performance benefit is most likely to be observed when there are multiple, relatively short data transfers for each communication step in an application.

In this paper, we present a benchmark designed to test such a communication pattern. The benchmark is based on the common "halo exchange" (or ghost-cell exchange) operation in applications that approximate the solution to partial differential equations. We compare a number of implementations of this benchmark with all three MPI RMA synchronization mechanisms:

`MPI_Win_fence`, the scalable synchronization (post/start/complete/wait), and passive target (`MPI_Win_lock/unlock`). For each of these mechanisms, MPI defines various parameters (assert options) that may be used by the programmer to help the MPI implementation optimize the operation. In addition, careful implementation can further improve performance [8].

Related Work. A number of papers have explored the performance of MPI RMA. The results in [4] focused on bandwidth for large messages and active-target synchronization in a variant of the ping-pong benchmark, though Table 1 there presents times for a single 4-byte message. The SKaMPI benchmark now supports tests of the MPI one-sided routines [1] and mentions a test similar to our halo test, but without considering varying numbers of neighbors or providing results. The MPI Benchmark Program Library [10] was developed to test the performance of MPI on the Earth Simulator and showed that MPI RMA was faster than the point-to-point operations on that system.

Evaluating the performance of MPI RMA requires careful attention to the semantics of the MPI RMA routines. The broadcast algorithms used in Appendix B and C of [6], for example, rely on `MPI_Get` being a blocking function, which it need not be. In implementations that take advantage of the nonblocking nature of `MPI_Get` allowed by the MPI Standard (for example, MPICH2 [8]), the code in Appendix B and C of [6] will indeed go into an infinite loop.

Papers that discuss the implementation of MPI RMA naturally include performance measurements; for example, see [2, 9]. The test we use in this paper is similar to Wallcraft's halo benchmark [11], but that benchmark does not use MPI one-sided communication and uses only four neighbors in the halo exchange. Wallcraft's halo benchmark has also been used in comparing MPI with other programming models [3].

2 The Benchmark

Our benchmark exchanges data with a selected number of partner processes. It mimics a halo, or ghost-cell, exchange that is a common component of parallel codes that solve partial differential equations. The code for this pattern, using MPI point-to-point communication, is as follows:

```
for (j=0; j<n_partners; j++) {
    MPI_Irecv( rbuffer[j], len, MPI_BYTE, partners[j], 0,
               MPI_COMM_WORLD, &req[j] );
    MPI_Isend( sbuffer[j], len, MPI_BYTE, partners[j], 0,
               MPI_COMM_WORLD, &req[n_partners+j] );
}
MPI_Waitall( 2*n_partners, req, MPI_STATUSES_IGNORE );
```

In the case of two partners, the neighbor processes are the processes with ranks one greater and one less than the rank of the process. In the case of four partners,

the neighbor processes mimic a two-dimensional decomposition. In the case of eight partners, the eight neighbors in a two-dimensional decomposition are used.

This test is chosen because it allows us to separate the data transfers (in the RMA case, the `MPI_Put` and `MPI_Get` calls) from the synchronization (e.g., the `MPI_Win_fence` call). It also better reflects the communication in many simulation applications than does the standard ping-pong test.

Each of the MPI RMA methods allows different options for optimization. For example, there are various "assert" options for `MPI_Win_fence`. Our test allows the selection of the following options.

Fence. This is active-target synchronization with `MPI_Win_fence`.

1. Allocate send and receive buffers with `MPI_Alloc_mem`.
2. Specify "no_locks" in `MPI_Win_create`.
3. Provide assert option `MPI_MODE_NOPRECEDE` on `MPI_Win_fence` before RMA calls and all of `MPI_MODE_NOSTORE`, `MPI_MODE_NOPUT`, and `MPI_MODE_NOSUCCEED` on the `MPI_Win_fence` after the RMA calls.

Post/Start/Complete/Wait. This is scalable active-target synchronization with `MPI_Win_post`, `MPI_Win_start`, `MPI_Win_complete`, and `MPI_Win_wait`.

1. Allocate send and receive buffers with `MPI_Alloc_mem`.
2. Specify "no_locks" in `MPI_Win_create`.

Passive. This is passive-target synchronization with `MPI_Win_lock` and `MPI_Win_unlock`.

1. Use locktype `MPI_LOCK_SHARED` (instead of `MPI_LOCK_EXCLUSIVE`).
2. Do not use a separate `MPI_Barrier` for the target processes to know that all RMA operations have completed. This is relevant for applications that may have an algorithmic reason for knowing that RMA operations are complete, such as a required `MPI_Allreduce`.

Since an implementation may require that only memory allocated with `MPI_Alloc_mem` be used for passive-target RMA, we do not attempt to use the passive-target mode without allocating memory in this way.

The testing methodology is the same as that used in `mpptest` and was described in [5]. It uses the minimum of an average time, where the time of an individual test (containing multiple iterations of the basic communication test) is large relative to the granularity and precision of the clock. Since the tests are implemented within the `mpptest` code, all of the many options for controlling message sizes and measurement details are available. The tests are available as part of the current distribution of `mpptest`, available at [7]. A script, `runhalo`, is provided that runs the tests with the various RMA optimization options.

To better understand the tests, we also measured the halo exchange as implemented with `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall` (as in the example code) and with persistent sends and receives. In addition, since `MPI_Win_fence` can be implemented with `MPI_Barrier` on cache-coherent SMPs where immediate direct-memory copy is used for the MPI RMA operations, we also measured the performance of `MPI_Barrier`.

3 Results

We ran our tests on a variety of platforms and with a variety of MPI implementations. Results for the native (vendor-supplied) implementations are provided for SGI Altix, Sun Fire, and IBM p655+ SMPs. We also include results for MPICH2 version 1.0.5 and Open MPI 1.2.0 on a Linux cluster.

The results of testing the performance-optimization features showed that only a few optimizations are exploited by the implementations we tested. Table 1 summarizes which optimization approaches provided a significant, measurable benefit in our tests. In the discussion of each platform, the results with the best choice of options are used.

Table 1. Optimizations that were observed to help in the halo tests. An "X" appears in the "All" row only if using multiple optimizations provides an improvement over a single optimization. "NA" means that the MPI implementation does not support that feature. No options provided a benefit on the IBM p655+.

Option	SGI Altix	SUN Fire	IBM p655+	MPICH2	Open MPI
AllocMem w Fence		X			
Nolocks w Fence					
Asserts w Fence				X	X
All w Fence					
AllocMem w PSCW	NA	X			
Nolocks w PSCW	NA				
All w PSCW	NA				
Shared locks	X	X			

The rest of this section describes the performance of the different RMA synchronization modes using the best set of optimization values. In the interests of space, we provide graphs for only a subset of our results, summarizing the measurements in the text.

3.1 SGI Altix

We ran our tests on three different SGI Altix SMP systems that are part of the Columbia supercomputer at the NASA Ames Research Center. These were the single-core SGI Altix 3700 and Altix 3700 Bx2 and the dual-core Altix 4700; the results in this paper are from the Altix 3700 Bx2. SGI's MPI implementation does not support the post/start/complete/wait method of synchronization, only fence and lock-unlock.

Figure 1 shows that the Altix has excellent RMA performance. Lock-put-unlock without an additional barrier performs significantly better than any other form of communication. For the 8-neighbors case, it is ten times faster than send-receive. Even the fence method for 8 neighbors (put-8) is more than twice as fast as send-receive.

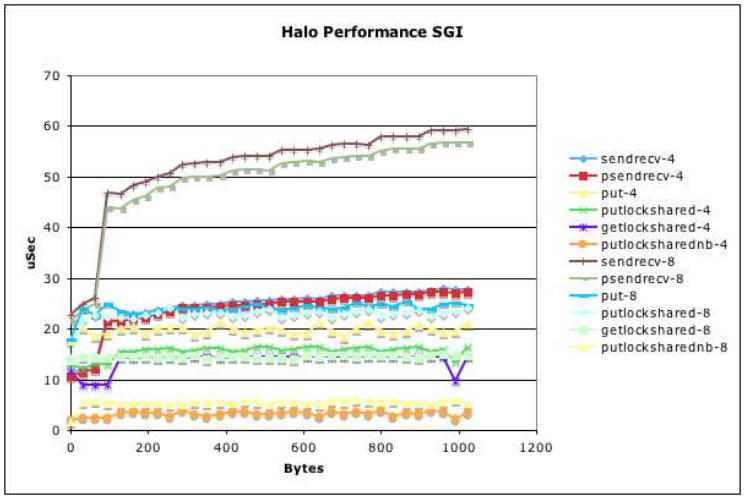

Fig. 1. Performance of halo exchange on SGI Altix with 16 processes. The best RMA results are compared with point-to-point; the legend indicates the number of neighbors (e.g., put-4 is put/fence with four neighbors, psendrecv-8 is persistent send/receive with eight neighbors, and nb stands for no barrier).

A surprising aspect of the Altix results is that the RMA optimization features in the MPI calls (e.g., the assert values in `MPI_Win_fence`) have no measurable effect, nor does using memory allocated with `MPI_Alloc_mem` (for fence). While this is attractive for the user (nothing to do), a closer look at all the data we collected suggests that additional optimizations could help in some cases. For example, in the two-neighbor case, put-fence was slower than send-recv by 50%. But since lock-unlock was significantly faster, a tuned version of fence that takes advantage of user-provided asserts should also be able to outperform send-recv.

3.2 Sun Fire

We ran our tests on the Sun Fire SMP cluster at the RWTH Aachen University using Sun's MPI. The specific machine we ran on was a Sun Fire E2900 with eight dual-core UltraSPARC IV 1.2 GHz CPUs. Figure 2 shows a subset of the results. As on the Altix, the performance of lock-unlock without an additional barrier is the best of all communication methods—it is twice as fast as send-receive. The performance of MPI RMA on this system is quite good if the memory used is allocated with `MPI_Alloc_mem`. The other optimization options had little or no effect on the performance of the halo tests. In particular, the `MPI_Win_fence` options had no effect. One unusual feature of this implementation is the extraordinarily long time required by `MPI_Alloc_mem` and `MPI_Win_create`. Times of

several seconds were measured; we rarely saw these routines take less than a few seconds when using 16 processes.[1]

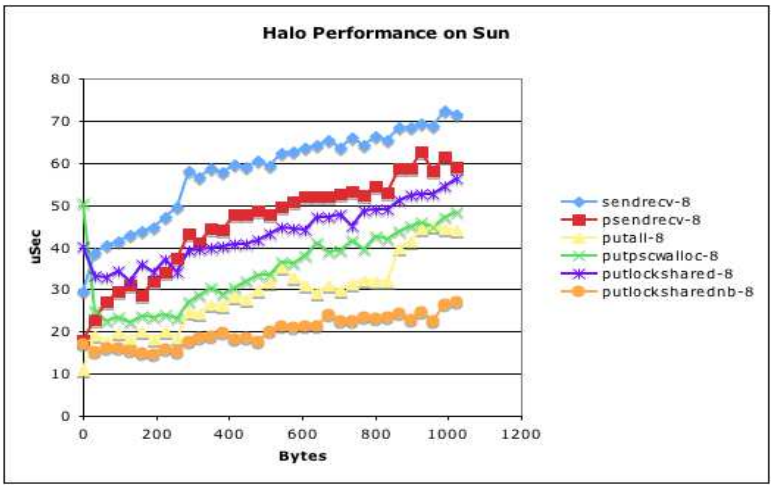

Fig. 2. Performance of 8-neighbor halo exchange on Sun Fire SMP with 16 processes in `MPI_COMM_WORLD`. putpscwalloc is the scalable synchronization with `MPI_Alloc_mem`. putlockshared is passive target with shared locks, and putlocksharednb omits the barrier that is necessary to ensure completion at the target.

3.3 IBM p655+

We ran our tests on the DataStar machine at the San Diego Supercomputer Center with IBM's MPI. The specific node we ran on was an IBM p655+ 8-way SMP. The p655+ has 1.7 GHz POWER4+ CPUs. Nodes in DataStar are connected with the Federation Switch; however, as our tests used a single node, the switch was not used.

With eight processes on an eight-node SMP, the RMA performance was very poor, on the order of forty times slower than the point-to-point performance. With seven processes on the same eight-node SMP, the RMA performance was still poor but an order of magnitude faster than with eight processes. This case is shown in Figure 3. The significant change in performance between eight and seven processes suggests that a thread is used for implementing the RMA operations and that the implementation is not prepared to handle the case where there are more threads than processors. To test this hypothesis, we also ran with four MPI processes on an eight-processor system. The performance in that case

[1] We were told that the performance problem with `MPI_Alloc_mem` has been fixed in Sun's ClusterTools 7; the version on the machine was ClusterTools 5.

was further improved over the seven-process case but was still poor relative to the point-to-point version. An `MPI_Barrier` on this system takes roughly 9 μsec on 8 processes, so the cost of a barrier or barrier-like synchronization is not a major contributor to the high cost of RMA on this system.

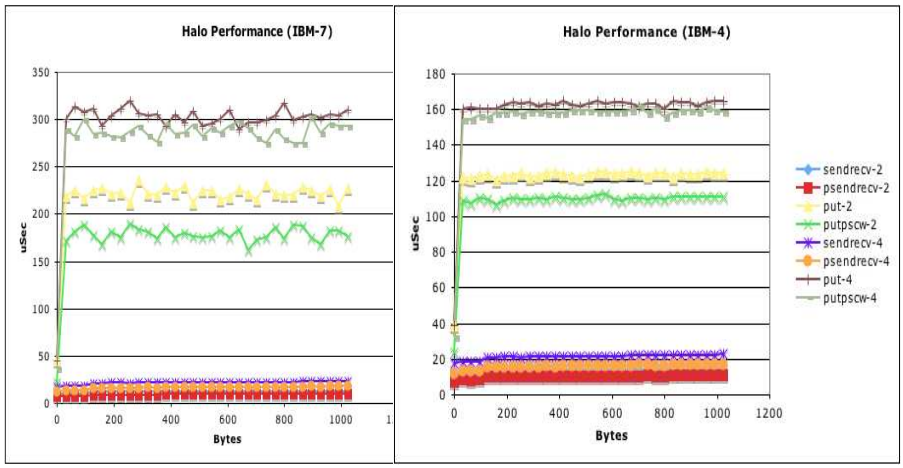

Fig. 3. Performance of RMA on IBM p655+. The chart to the left is with 7 processes on an 8-node SMP; to the right is 4 processes on an 8-node SMP. Results for two and four neighbors are shown using the two active target synchronization methods (put and putpscw in the legend) and point to point with nonblocking and persistent send/receive (sendrecv and psendrecv in the legend, respectively).

3.4 Linux Cluster

We also ran the tests on the Jazz cluster at Argonne, which has 2.4 GHz Pentium Xeon nodes and both a Myrinet 2000 and 100 Mb/s Ethernet interconnect. We used two MPI implementations, MPICH2 1.0.5 and Open MPI 1.2.0. The cluster uses an older version of the native GM library for Myrinet, and we could not build Open MPI for that version. Hence we used TCP over Myrinet for communication with both MPICH2 and Open MPI. As the results in Figure 4 show, the best performance was achieved with the point-to-point operations for both implementations. The reason is that in the absence of hardware and software support for RMA from the network-transport layer, the MPI RMA operations are simply implemented on top of lower-level point-to-point operations. Nonetheless, RMA with MPICH2 performs significantly better than with Open MPI. Some of this performance improvement is due to the optimizations in MPICH2 that minimize the synchronization overhead associated with MPI RMA [8].

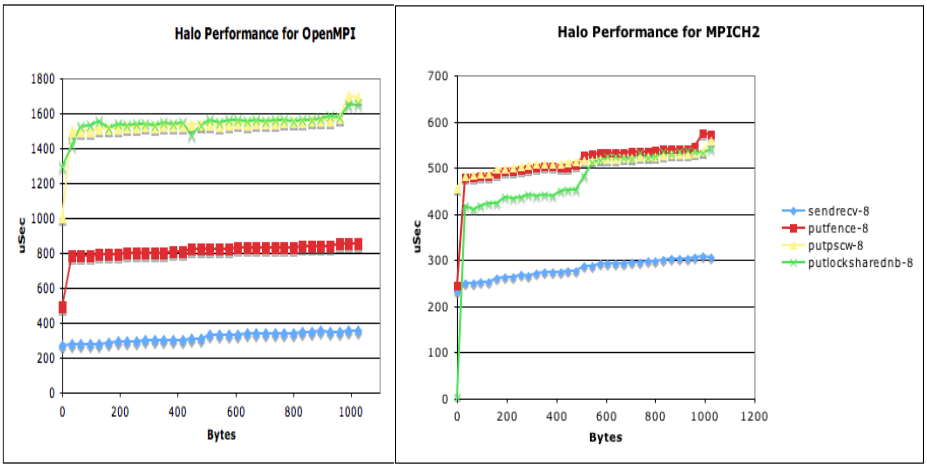

Fig. 4. Performance of 8-neighbor halo exchange with 16 processes on the Linux cluster by using Open MPI (left) and MPICH2 (right).

4 Conclusions

We have shown that implementations of MPI RMA can provide a performance advantage on systems with hardware support for remote-memory operations, particularly when there are multiple RMA operations per synchronization operation. The SGI Altix and Sun Fire provided surprisingly good performance for the passive-target RMA operations; in fact, the performance was so good that it may be possible to improve the performance of the active-target RMA methods by making use of the approach used for the passive-target RMA.

We measured surprisingly poor performance on an IBM SMP. We suspect that the implementation is not optimized for MPI RMA operations and relies on separate threads that may be running in a polling mode, thus leading to very poor performance when there are fewer processors than at least two times the number of MPI processes.

Few of the flags provided by the MPI standard are exploited by the implementations. This situation was reflected in the surprisingly high overhead for active-target RMA operations on most of the platforms. We hope that our benchmark will encourage MPI implementors to exploit these features.

Acknowledgments

We thank the RWTH Aachen University, NASA Ames, and the San Diego Supercomputer Center for providing computing time on their systems. We particularly thank Subhash Saini and Dale Talcott for running the tests on the Altix machines and Anthony Chan for running the tests on the Linux cluster.

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 DE-AC02-06CH11357.

References

1. Werner Augustin, Marc-Oliver Straub, and Thomas Worsch. Benchmarking one-sided communication with SKaMPI 5. In Beniamino Di Martino, Dieter Kranzlmüller, and Jack Dongarra, editors, *PVM/MPI*, volume 3666 of *Lecture Notes in Computer Science*, pages 301–308. Springer, 2005.
2. S. Booth and E. Mourão. Single sided MPI implementations for SUN MPI. In *Proceedings of Supercomputing'2000 (CD-ROM)*, Dallas, TX, November 2000. IEEE and ACM SIGARCH. EPCC, The University of Edinburgh.
3. Co-Array Fortran vs MPI. `http://www.co-array.org/cafvsmpi.htm`.
4. Edgar Gabriel, Graham E. Fagg, and Jack Dongarra. Evaluating the performance of MPI-2 dynamic communicators and one-sided communication. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *PVM/MPI*, volume 2840 of *Lecture Notes in Computer Science*, pages 88–97. Springer, 2003.
5. William D. Gropp and Ewing Lusk. Reproducible measurements of MPI performance characteristics. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 11–18. Springer Verlag, 1999.
6. Glenn R. Luecke, Silvia Spanoyannis, and Marina Kraeva. The performance and scalability of SHMEM and MPI-2 one-sided routines on a SGI Origin 2000 and a Cray T3E-600. *Concurrency and Computation: Practice and Experience*, 16(10):1037–1060, 2004.
7. MPPTEST - Measuring MPI Performance. `http://www.mcs.anl.gov/mpi/mpptest`.
8. Rajeev Thakur, William Gropp, and Brian Toonen. Optimizing the synchronization operations in MPI one-sided communication. *International Journal of High-Performance Computing Applications*, 19(2):119–128, Summer 2005.
9. Jesper Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Proceedings of Supercomputing'2000 (CD-ROM)*, Dallas, TX, November 2000. IEEE and ACM SIGARCH. NEC Europe Ltd.
10. Hitoshi Uehara, Masanori Tamura, and Mitsuo Yokokawa. An MPI benchmark program library and its application to the Earth Simulator. In *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing*, pages 219–230, London, UK, 2002. Springer-Verlag.
11. Alan J. Wallcraft. SPMD OpenMP versus MPI for ocean models. *Concurrency: Practice and Experience*, 12(12):1155–1164, 2000.