Understanding the Behavior and Performance of Non-blocking Communications in MPI*

Taher Saif and Manish Parashar

The Applied Software Systems Laboratory
Department of Electrical and Computer Engineering
Rutgers University, Piscataway, NJ 08854, USA
{taher,parashar}@caip.rutgers.edu

Abstract. The behavior and performance of MPI non-blocking message passing operations are sensitive to implementation specifics as they are heavily dependant on available system level buffers. In this paper we investigate the behavior of non-blocking communication primitives provided by popular MPI implementations and propose strategies for these primitives than can reduce processor synchronization overheads. We also demonstrate the improvements in the performance of a parallel Structured Adaptive Mesh Refinement (SAMR) application using these strategies.

1 Introduction

The Message Passing Interface (MPI) [1] has evolved as the de-facto message passing standard for supporting portable parallel applications - with commercial as well as public-domain implementations available for most existing platforms including general purpose clusters to high-performance systems such as IBM SP.

An important design goal of the MPI standard is to allow implementations on machines with varying characteristics. For example, rather than specifying how operations take place, the MPI standard only specifies what operations do logically. Consequently, MPI can be easily implemented on systems that buffer messages at the sender, receiver, or do no buffering at all. It is typically left to the vendors to implement MPI operations in the most efficient way as long as their behavior conforms to the standards. As a result of this, MPI implementations on different machines often have varying performance characteristics that are highly dependant on factors such as implementation design, available hardware/operating system support and the sizes of the system buffers used.

The behavior and performance of MPI non-blocking message passing operations are particularly sensitive to implementation specifics as they are heavily dependant on available system level buffers and other resources. As a result,

* The work presented here was supported in part by the National Science Foundation via grants numbers ACI 9984357 (CAREERS), EIA 0103674 (NGS) and EIA 0120934 (ITR), and by DOE ASCI/ASAP (Caltech) via grant number PC295251.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 173–182, 2004.

naive use of these operations without an understanding of the underlying implementation can result in serious performance degradations, often producing synchronous behaviors.

We believe that an efficient and scalable use of MPI non-blocking communication primitives requires an understanding of their implementation and its implication on application performance. This paper has two key objectives: (1) To investigate and understand the behavior of non-blocking communication primitives provided by two popular MPI implementations: the public domain MPICH [2] implementation on a Linux cluster, and the proprietary IBM implementation on an IBM SP2 [3]. (2) To propose and evaluate usage strategies for these primitives that the parallel programmer can implement to reduce processor synchronization and optimize application performance. We use the proposed strategies to optimize the performance of parallel implementations of scientific/engineering simulations that use finite difference methods on structured adaptive meshes [4].

2 Non-blocking MPI: Behavior and Performance

The generic operation of a non-blocking MPI communication is a "three step" process in which the implementation (of the non-blocking communication) decouples the send and receive operations by using system and/or application buffers at the sender and receiver processes, allowing computation and communication to be overlapped. However, this decoupling is strictly limited by the size of the buffers available to copy the message. MPI implementations typically switch to a synchronous communication mode when the message size exceeds the available buffer size, where the sender waits for an acknowledgement from the receive side before sending out the data.

In this section we experimentally investigate the behavior and performance of non-blocking MPI communications in two popular MPI implementations: MPICH on a Beowulf cluster, and IBM MPI on the IBM SP2.

The test kernel used for these experiments - as illustrated in Figure 1 - is a typical non-blocking communication implementation between two processes in which one sends and the other receives. In this kernel the sending process (process 0) issues MPI_Isend (*IS*) at time-step T0 to initiate a non-blocking send operation while the receiving process (process 1) posts a matching MPI_Irecv (*IR*) call. Both processes then execute unrelated computation before executing an MPI_Wait call at T3 to wait for completion of the communication. In the following discussion we denote MPI_Wait posted on the send side as *Ws* and the MPI_Wait posted on the receive side as *Wr*. The processes synchronize at the beginning of the kernel and use deterministic offsets to vary values of T0, T1, T2 and T3 at each process. For each configuration (value of T0, T1, T2 and T3 at each process) we conducted a number of experiments varying the message size, system buffer size and number of messages exchanged. The objectives of these experiments included determining thresholds at which the non-blocking calls synchronize, the semantics of synchronization once this threshold is reached, and possibility of deadlocks.

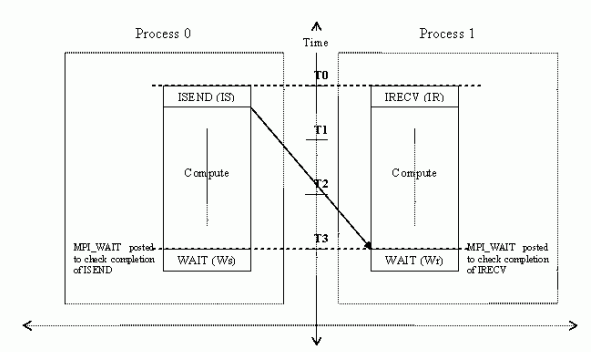

Fig. 1. Operation of the test kernel used in the experimental investigation.

2.1 MPICH on a Linux Beowulf Cluster

The first MPI implementation analyzed is MPICH version 1.2.5, release date January 6, 2003 [3] on Frea, a 64 node Linux Beowulf SMP cluster at Rutgers University. Each node of cluster has a 1.7 GHz Pentium 4 processor with 512 MB main memory. The MPICH profiling tool Upshot [14] is used for the profiles and timing graphs presented below.

Our first experiment investigates the effect of message size on non-blocking communication semantics. In this experiment the value of T0 - T3 are approximately the same on the two processes, and the message size was varied. The system buffer size was maintained at the default value of 16K. For smaller message sizes (1KB), we observe that *IS* and *IR* return without blocking (Figure 2). Furthermore, *Ws* and *Wr*, posted after local computations, return almost immediately, indicating complete overlap. However, for message sizes greater than or equal to 60 KB, *IS* blocks and returns only when the receiver process posts *Wr* (Figure 3). We can further see from the Figure that *Wr* blocks until the message delivery completes. This threshold is dependent on the system buffer size as discussed below.

To further understand the synchronizing behavior of MPI_Isend for large message sizes, we modified our experiment to post a matching MPI_Test (a non-blocking variant of MPI_Wait) on the receiver side (i.e. process 1) in the middle of the computation phase. As shown in Figure 4, in this case MPI_Isend returns as soon as MPI_Test is posted. It was also seen that the *Wr* posted after com-

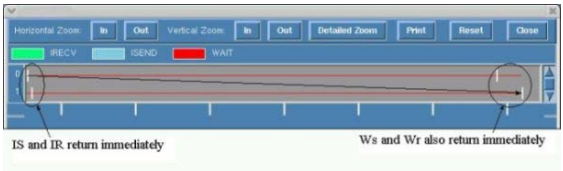

Fig. 2. Profile of the test on MPICH where process 0 (top) sends a 1 KB message to process 1 (bottom).

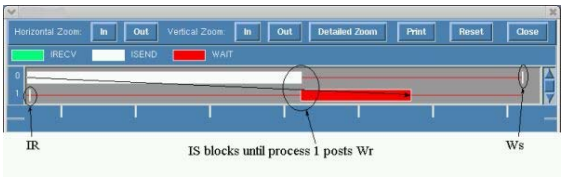

Fig. 3. Profile of the test on MPICH where process 0 (top) sends a 60 KB message to process 1 (bottom).

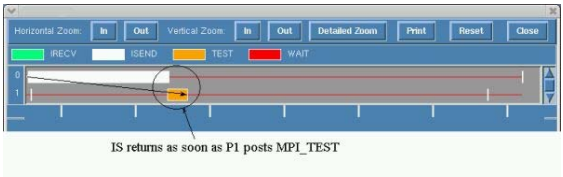

Fig. 4. Profile of the test on MPICH in which process 1 (bottom) posts an intermediate MPI_Test.

putation returns almost immediately, indicating that the message was already delivered during the computation. This indicates that MPI_Isend blocks for large messages until the completion of the corresponding MPI_Irecv is checked using either blocking (MPI_Wait) or non-blocking (MPI_Test). Note that, as the MPI implementation optimizes the number of messages sent to a single destination, the message size threshold is cumulative. That is, in the above case MPI_Isend switches to blocking semantics when the cumulative size of outstanding messages to a particular process is 60KB. For example, when we repeated the test using 3 sends of size 20KB each (instead of one of size 60KB), the same non-blocking behavior was observed.

The experiment plotted in Figure 5 evaluates potential deadlocks if two processes simultaneously send large messages to each other using MPI_Isend and block. The Figure shows that process 1 initially blocks but then returns after a certain time instead of waiting for process 0 to post a *Wr*. Process 0 however blocks until *Wr* is posted on process 1. This behavior seems to indicate a non-deterministic time out mechanism to ensure progress.

In the case of the Frea Beowulf cluster, default TCP socket buffer size is 16KB. This can be increased by either using the environment variable P4_SOCKBUFSIZE or using the command line option -p4sctrl bufsize=<size>.

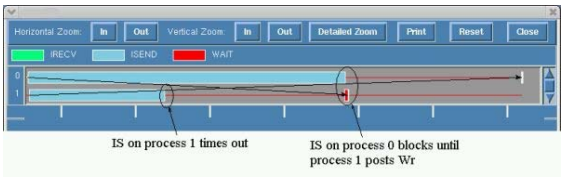

Fig. 5. Profile of the test on MPICH in which both processes post MPI_Isend. Message size is 60Kb.

We repeated the experiments using messages of size 60KB but increased the TCP socket buffer size. We observed that for a TCP socket buffer size of 64KB, IS did not block.

Analysis and Discussion. To try to understand the behavior of the MPICH non-blocking communication presented above let us consider its implementation. MPICH has a layered software architecture [5] consisting of (a) a high level MPI API layer (b) a middle Abstract Device Interface (ADI) layer and, (c) a D1evice layer. The Device layer defines three protocols to send messages based on the message size: short, eager (long) and rendezvous (very long). In the case of relatively short messages, for example 1KB, MPICH can copy the message directly into the system socket buffer and thus send the message out onto the network using the eager protocol, allowing MPI_Isend to return immediately.

In the case of larger messages (e.g. 60KB), the system socket buffer is not large enough to accommodate the message and MPICH cannot directly copy the message into the buffer. Instead, it switches to the rendezvous protocol, which requires the sending process to synchronize with the receiving process before the message is sent out. As a result MPI_Isend, which should return immediately irrespective of the completion mode, now has to wait for the corresponding *Wr* for an acknowledgement(Figure 3). Similarly, when a matching MPI_Test is posted at the receiver process, it essentially sends an acknowledgement back to the sender which caused the blocked MPI_Isend to return. When the TCP/IP socket buffer size is increased to 64 KB, MPICH can copy the 60 KB message directly into the socket buffer and use the eager protocol allowing the MPI_Isend call to return without blocking.

Finally, due to MPICH optimizations, the blocking behavior of MPI_Isend depends on the system socket buffer and the cumulative size of the outstanding messages rather than the actual number of messages sent. Consequently, reducing the message size by breaking up the total message into smaller messages will not yield any performance improvement.

Optimization Strategies. Based on the analysis presented above we identify two strategies to address the blocking behavior of MPI_Isend in MPICH. The first strategy is obvious, increase the TCP socket buffer size. However this option is not scalable since the total buffer space grows with the number of processes. Further, every system imposes a hard limit on the total socket buffer size. As a result this option has only limited benefits and any further optimization must be achieved at the applications level.

It is clear from the analysis presented above that the only way to prevent MPI_Isend from blocking is for the receiving process to return an acknowledgement using a (blocking or non-blocking) test for completion call. Our second strategy is to use calls to the non-blocking test for completion (MPI_Test or its variant) on the receive side to release a blocked sender.

To illustrate this consider the code snippet (Figure 6) for a typical loose-synchronous application, for example, a finite-difference PDE solver using ghost

```
for m=1 to number_of_messages_to_receive {
        MPI_IRECV(m, recv_msgid_m)
}
***COMPUTE***
for n=1 to number_of_messages_to_send{
        MPI_ISEND(n, send_msgid_n)
        MPI_WAIT(send_msgid_n)
}
***COMPUTE***
MPI_WAITALL(recv_msgid_*)
```

Fig. 6. MPICH: Unoptimized algorithm.

```
for m=1 to number_of_messages_to_receive{
        MPI_IRECV(m, recv_msgid_m)
}
***COMPUTE***
for n=1 to number_of_messages_to_send{
        MPI_ISEND(n, send_msgid_n)
        MPI_WAIT(send_msgid_n)
}
MPI_TESTALL(recv_msgid_*)
***COMPUTE***
MPI_WAITALL(recv_msgid_*)
```

Fig. 7. MPICH: Optimized algorithm.

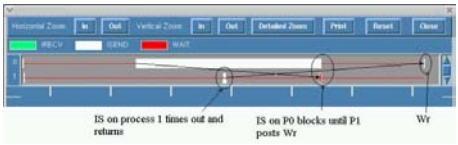

Fig. 8. MPICH: Unoptimized algorithm.

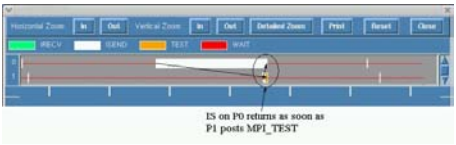

Fig. 9. MPICH: Optimized algorithm.

communications. In this pseudo-code, each process posts non-blocking receive calls before computing on its local region of the grid. After finishing computation, it then sends its data to update the ghost regions of its neighboring processors using the MPI_Isend/MPI_Wait pair. The process may do some further local computation and then finally waits to update it own ghost regions, possibly using an MPI_Waitall. In this case, if the message size is greater than 60KB the MPI_Isend will block until the corresponding MPI_Waitall is called on the receiving process as shown in Figure 8.

If we now insert an intermediate MPI_Testall call as shown in Figure 7, MPI_Isend returns as soon as the receiver posts the test (Figure 9). While the MPI_Testall call does have a cost, this cost is small compared to the performance gain.

2.2 IBM MPI on the SP2

The second MPI implementation analyzed is the IBM native implementation (version 3 release 2) [3] on the IBM SP2, BlueHorizon, a teraflop-scale Power3 based clustered SMP system at the San Diego Supercomputing Center. The machine consists of 1152 processors, each having 512 GB of main memory. Once again, our first experiment investigates the effect of message size on non-blocking communication semantics. For smaller message sizes (1KB), we observe the expected non-blocking semantics. This is also true for larger messages sizes (greater than 100 KB) as shown in Figure 10.

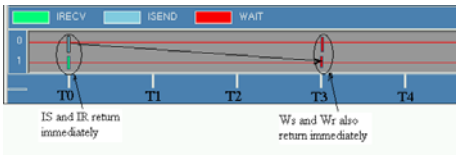

Fig. 10. SP2: *Ws* and *Wr* posted at the same time-step.

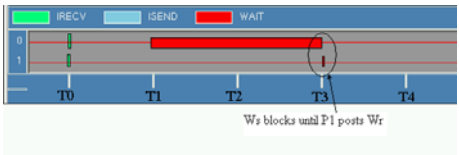

Fig. 11. SP2: *Ws* and *Wr* posted at different time-steps.

To further understand the effect of increasing message size on the behavior of non-blocking communications in the IBM MPI, we moved *Ws* to T1, i.e. directly after the send to simulate the situation where one might want to reuse the send buffer. *Wr* remained at T3. In this case, for message sizes greater than or equal to 100KB, *Ws* blocked until *Wr* was posted by the receiver at T3 (Figure 11).

In an experiment where both processes exchange messages, *IS* and *IR* are posted at T0, process 0 posts *Ws* at T1 while process 1 posts *Ws* at T2, and both processes post *Wr* at T3. The message size is maintained at 100KB. In this case deadlock is avoided in an interesting manner since *Ws*, posted at T1 and blocks on process 0, returns as soon as process 1 posts *Ws* at T2, rather than waiting for the corresponding *Wr* on T3.

Analysis and Discussion. The SP2 parallel environment imposes a limit (called the eager limit) on the total message size that can be sent out asynchronously. When message sizes exceed this limit, the IBM MPI implementation switches to a synchronous mode. However, in this case, it is the *Ws* call that blocks until an acknowledgement is received from the receiver process. Consequently in the experiment above, *Ws* blocks until *Wr* is posted at the receiving process. The analysis above also shows that the synchronization call on the receive side need not be a matching wait. In fact the receiver may post any call to MPI_Wait (or any of its variants) to complete the required synchronization.

Optimizations Strategies. The POE users' guide [3] specifies the environment variable, MP_EAGER_LIMIT, which defines the size of MPI messages that can be sent asynchronously. However, as the number of processes increase, trying to increase MP_EAGER_LIMIT simply reduces the amount of memory available to the application.

A more scalable strategy is to address this at the application level by appropriatly positioning *IS*, *IR*, *Ws* and *Wr* calls. The basic strategy consists of delaying *Ws* until after *Wr* and is illustrated in Figures 12 and 13.

To illustrate the strategy, consider a scenario in which two processes exchange a sequence of messages and the execution sequence is split into steps T0-T3. Both processes post MPI_Irecv (*IR*) calls at T0 and Wall denotes a MPI_Waitall call. Assume that, due to load imbalance, process 0 performs computation until T2 while process 1 computes only till t1. *Ws* posted on process 1 at T1 will block until process 0 posts *Ws* at T2. For a large number of messages, this delay can

```
for n=1 to number_of_messages_to_receive{
        MPI_IRECV(n, msgid_n)
}
***COMPUTE***
for n=1 to number_of_messages_to_send{
        MPI_ISEND(n, send_msgid_n)
        MPI_WAIT(send_msgid_n)
}
MPI_WAITALL(recv_msgid_*)
```

Fig. 12. SP2: Unoptimized algorithm.

```
for n=1 to number_of_messages_to_receive{
        MPI_IRECV(n, msgid_n)
}
***COMPUTE***
for n=1 to number_of_messages_to_send{
        MPI_ISEND(n, send_msgid_n)
}
MPI_WAITALL(recv_msgid_*+send_msgid*)
```

Fig. 13. SP2: Optimized algorithm.

become quite significant. Consequently, to minimize the blocking overhead due to *Ws* on process 1, it must be moved as close to T2 as possible. Now, if *Ws* is removed from the send loop and a collective MPI_Waitall is posted as shown in Figure 13, it is observed that process reaches T2, it has already posted *IS* for all of its messages and is waiting on Wall, thus reducing synchronization delays.

3 Evaluation of Communication Performance in SAMR

Dynamic Structured Adaptive Mesh Refinement (SAMR) techniques [4] for solving partial differential equations provide a means for concentrating computational effort to appropriate regions in the computational domain. These methods (based on finite differences) start with a base coarse grid with minimum acceptable resolution that covers the entire computational domain. As the solution progresses, regions in the domain requiring additional resolution are recursively tagged and finer grids are laid over these tagged regions of the coarse grid [4].

Parallel implementations of hierarchical SAMR applications typically partition the adaptive heterogeneous grid hierarchy across available processors, and each processor operates on its local portions of this domain in parallel [7]. Due to their irregular load distributions and communication requirements across levels of the grid hierarchy, parallel SAMR applications make extensive use of non-blocking MPI primitives so as to overlap intra-level communications with computations on the interior region.

A typical implementation of intra-level communications in parallel SAMR applications is similar to the ghost communication associated with parallel finite difference PDE solvers as described in Section 2. Clearly, the optimizations proposed by us in Section 2 can be applied here to reduce the synchronization costs.

Evaluation Using the RM3D Kernel. To evaluate the impact of the proposed optimization strategies on application performance we used the 3-D version of the compressible turbulence application kernel (RM3D) which uses SAMR

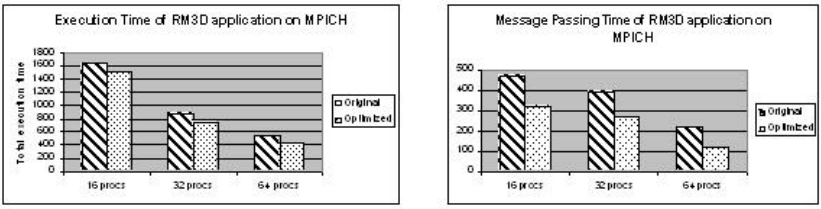

Fig. 14. Comparison of execution and communication times on Frea (MPICH).

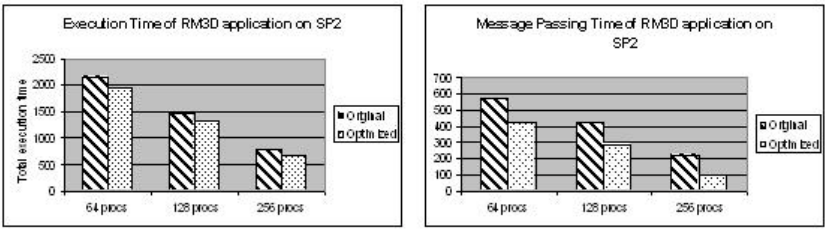

Fig. 15. Comparison of execution and communication times on SP2 (IBM POE).

techniques to solve the Richtmyer-Meshkov instability [9]. The experiments consist of measuring the message passing and application execution times for the RM3D application kernel before and after incorporating our optimizations strategies outlined in this paper, on both Frea and Blue Horizon. Except for the optimizations in the message passing algorithm, all other application-specific and refinement-specific parameters are kept constant. The results of the evaluation for MPICH on Frea for 16, 32 and 64 processors are shown in Figure 14. These runs used a base grid size of 128*32*32 and executed 100 iterations. We observe that the reduction in communication time is approximately 27%.

On the SP2 the evaluation run used a base grid size of 256*64*64 and executed 100 iterations. Figure 15 shows the comparisons of the execution times and communication times respectively for 64, 128 and 256 processors. In this case we observe that the reduction in communication time is approximately 44%.

4 Summary and Conclusions

In this paper we experimentally analyzed the behavior and performance of non-blocking communication provided by two popular MPI implementations. It is important to note that the blocking behavior described by us is not a bug in the message passing softwares. Rather it is due to inherent limitations in the underlying hardware architectures.

We used the strategies proposed in this paper to optimize the performance of the SAMR-based Richtmyer-Meshkov compressible turbulence kernel. Our evaluation shows that the proposed strategies improved the performance by an average of approximately 27% for MPICH and 44% for IBM MPI.

References

1. The MPI Forum. The MPI Message-Passing Interface Standard 2.0. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html, September 2001.
2. MPICH - A Portable MPI Implementation. http://www-unix.mcs.anl.gov/mpi/mpich/
3. Parallel Environment (PE) for AIX V3R2.0: Operation and Use, Vol. 1. December, 2001.
4. M. Berger and J. Oliger, "Adaptive Mesh Refinement for Hyperbolic partial Differential Equations", *Journal of Computational Physics*, Vol. 53, pp. 484-512, 1984.
5. W. Gropp, E. Lusk, A. Skjellum and N. Doss, "MPICH: A High-Performance, Portable Implementation for MPI Message-Passing Interface", *Parallel Computing*, 22, 1996, pp. 789-828.
6. W. Gropp and E. Lusk, "MPICH Working Note: The Second-Generation ADI for the MPICH Implementation of MPI", http://www-unix.mcs.anl.gov/mpi/mpich/, 1996.
7. M. Parashar and J. C. Browne, "On Partitioning Dynamic Adaptive Grid Hierarchies", Proceedings of the *29th Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, IEEE Computer Society Press, pp. 604-613, January 1996.
8. V. Herrarte and E. Lusk, "Studying Parallel Program Behavior with upshot", Technical Report ANL-91/15, Argonne National Laboratory, 1991.
9. J. Cummings, M. Aivazis, R. Samtaney, R. Radovitzky, S. Mauch, and D. Meiron, "A virtual test facility for the simulation of dynamic response in materials" *Journal of Supercomputing*, 23:39-50, 2002.