ARCHITECTURE SPECIFIC COMMUNICATION OPTIMIZATIONS FOR STRUCTURED ADAPTIVE MESH-REFINEMENT APPLICATIONS

BY TAHER SAIF

A thesis submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Professor Manish Parashar

and approved by

____

____

____

New Brunswick, New Jersey

January, 2004

ABSTRACT OF THE THESIS

Architecture Specific Communication Optimizations for Structured Adaptive Mesh-Refinement Applications

by Taher Saif

Thesis Director: Professor Manish Parashar

Dynamic Structured Adaptive Mesh Refinement (SAMR) techniques for solving partial differential equations provide a means for concentrating computational effort to appropriate regions in the computational domain. Parallel implementations of these techniques typically partition the adaptive heterogeneous grid hierarchy across available processors, and each processor operates on its local portions of this domain in parallel. However, configuring and managing the execution of these applications presents significant challenges in resource allocation, data-distribution and load balancing, communication and coordination, and runtime management. 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 to reduce synchronization overheads.

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 resources. As a result, naive use of these operations without an understanding of architectural constraints and the underlying MPI 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 on the specific architecture and its implication on application performance.

Specifically this thesis makes the following contributions:

- It investigates and understands the behavior of non-blocking communication primitives provided by popular MPI implementations on Linux and IBM SP2 clusters, and proposes usage strategies for these primitives than can reduce processor synchronization and optimize SAMR application performance.
- It proposes a design for multi-threaded communication algorithms for future architectures like IBM Blue Gene/L which use separate hardware for computation and communication

Acknowledgements

I would like to thank my research advisor Dr. Manish Parashar for his invaluable guidance, support and encouragement during the course of this work and throughout my graduate studies at Rutgers. I am thankful to Dr. Deborah Silver and Dr. Yanyong Zhang for being on my thesis committee and for their advice and suggestions regarding this thesis. I would like to thank Sumir Chandra for valuable research discussions and cooperation related to this work. I am grateful to the wonderful CAIP computer facilities staff for their assistance and support. I would like to thank my colleagues at the Applied Software Systems Laboratory (TASSL) for their help and for creating a wonderful environment which makes research a lot of fun. I am thankful to all my friends at Rutgers for making my stay a memorable one. Finally, I would like to thank my family back in India and here in United States without whose love and support this would not have been possible.

Dedication

To my Parents

Table of Contents

Abstract ii

Acknowledgements iv

Dedication v

List of Figures viii

1. Introduction 1

1.1. Contributions 3

1.2. Organization 4

2. Message Passing in MPI 5

2.1. Overview of point to point communication 5

2.2. Asynchronous non-blocking MPI communication 6

3. Understanding the Behavior and Performance of Non-blocking MPI Communications 9

3.1. Test Kernel 9

3.2. MPICH on a Linux Beowulf Cluster 10

3.2.1. Behavior of Non-Blocking Primitives 10

3.2.2. Analysis and Discussion 13

3.2.3. Optimization Strategies for Non-blocking communications in MPICH 14

3.3. The Parallel Operating Environment (POE) on IBM SP2 17

3.3.1. Behavior of Non-Blocking Primitives 17

3.3.2. Analysis and Discussion 19

3.3.3. Optimizations Strategies for Non-blocking Communications in the IBM POE . . . 20

4. Evaluation of communication performance in SAMR Applications 22

4.1. SAMR Overview . . . 22

4.2. Communication Requirements of Parallel SAMR Applications . . . 24

4.2.1. Communication overheads in parallel SAMR applications . . . 25

4.3. Grid Adaptive Computational Engine . . . 25

4.3.1. Intra-level communication model . . . 26

4.3.2. Optimizations . . . 27

4.4. Evaluation using the RM3D SAMR Kernel . . . 29

5. Future work - Design of an independent multi-threaded communication engine for parallel hardware architectures . . . 31

5.1. Introduction . . . 31

5.2. Algorithm . . . 32

5.3. Future Work . . . 34

6. Conclusions . . . 35

References . . . 37

List of Figures

2.1. Schematic of a generic implementation of MPI non-blocking communications. . . . 7
3.1. Operation of the test kernel used in the experimental investigation. . . 10
3.2. Profile of the test on MPICH where process 0 (top) sends a 1 KB message to process 1 (bottom). . . . 11
3.3. Profile of the test on MPICH where process 0 (top) sends a 60 KB message to process 1 (bottom). . . . 11
3.4. Profile of the test on MPICH in which buffered mode send is used. . . 12
3.5. Profile of the test on MPICH in which process 1 (bottom) posts an intermediate MPI_Test 12
3.6. Profile of the test on MPICH in which both processes post MPI_Isend. Message size is 60Kb. 13
3.7. Example of a non-blocking implementation on MPICH 15
3.8. Optimization of the non-blocking implementation on MPICH 16
3.9. Example of a non-blocking implementation on MPICH 16
3.10. Optimization of the non-blocking implementation on MPICH. 17
3.11. Profile of the test on SP2 where Ws and Wr are posted at the same time-step. 18
3.12. Profile of the test on SP2 where Ws and Wr are posted at different time-steps. 18
3.13. Profile of the test on SP2 showing the completion semantics of Ws. . 18
3.14. Example of a non-blocking implementation on SP2. 20
3.15. Optimization of the non-blocking implementation on SP2. 21
4.1. Grid Hierarchy. 23

4.2. Intra-level Communication model in a parallel SAMR application framework. 26
4.3. Staggered Sends - Optimizing SAMR Intra-level Communication for MPICH. 27
4.4. Delayed Waits - Optimizing SAMR Intra-level Communication for IBM POE. 28
4.5. Comparison of execution and communication times on Frea (MPICH) 30
4.6. Comparison of execution and communication times on SP2 (IBM POE) 30
5.1. Schematic of a multi-threaded algorithm for SAMR applications . . . 33

Chapter 1
Introduction

Ever since the invention of the computer, users have demanded more and more computational power to tackle increasingly complex problems. In particular, the numerical simulation of scientific and engineering problems creates an insatiable demand for computational resources, as researchers seek to solve larger problems with greater accuracy and in a shorter time. An example of such a problem are applications that use Structured Adaptive Mesh Refinement (SAMR) techniques for solving partial differential equations [1]. 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 tagged and finer grids are laid over these tagged regions of the coarse grid. The accuracy of these techniques is determined by the resolution that can be obtained on finer grids and the amount of computing power available to complete the calculations in a timely manner.

A common means of increasing the amount of computational power available for solving a problem is to use parallel computing. A parallel computer consists of two or more independent processors connected by some form of communication network. If a problem can be sub-divided into n smaller problems, a parallel program can be written to concurrently solve those sub-problems on n different processors. Ideally this would take $1/n^{th}$ of the time that would be required to solve the same problem using one processor. But this is rarely the case for two main reasons. Firstly, many problems contain significant amounts of computation that cannot be parallelized easily at all. Secondly, many problems require significant amounts of communication and synchronization between sub-problems, which can introduce long delays.

There are a number of different methodologies that can be used to create parallel

programs that run on parallel computers [4]. Two of the most fundamental methodologies are closely associated with the main parallel computer architectures that are available. The first is shared memory programming for tightly coupled *shared-memory* multiprocessors which use a single address space. Systems based on this concept allow processor communication through variables stored in a shared address space. The second major methodology for parallel programming, one which we will be concentrating on in this thesis, employs a scheme by which each processor has its own local memory. In such *distributed-memory* multiprocessor systems (also called clusters); processors communicate by explicitly copying data from one processor's memory to another using *message passing*.

The message passing programming model has gained wide use for developing high performance parallel and distributed applications. In this model, processes explicitly pass messages to exchange information. As a result, it is naturally suited to multi-computer systems without shared address spaces. Communications can be synchronous (a send (receive) operation completes only when a matching receive (send) is executed) or asynchronous (send and receive operations can complete independently). The latter communication semantics are particularly useful for overlapping communication and computations, hiding latencies and supporting applications with irregular loads and communication patterns. Higher level message passing operations support barriers, many-to-one and many-to-many communications and reductions.

The Message Passing Interface (MPI) - a library of functions (in C) or subroutines (in Fortran) that you insert into source code to perform data communication between processes - has evolved as the de-facto message passing standard for supporting portable parallel applications. Commercial as well as public-domain implementations of the MPI specification proposed by the MPI Forum [14] are available for most existing parallel platforms including clusters of networked workstations running general purpose operating systems (e.g. Linux, Windows) and high-performance systems such as IBM SP, SGI Power Challenge and CRAY T3E.

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. Consequently, the performance of an MPI call usually depends on how it is actually implemented on the system.

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, naive use of these operations without an understanding of the underlying implementation can result in serious performance degradations, often producing synchronous behaviors. For example, in [11] White et al describe experiments that illustrate the degree to which asynchronous communication primitives in popular MPI implementations support overlap on different parallel architectures, and the degree to which programming for overlap using these primitives can improve actual application performance. They also enumerate environment variables that improve the performance of the MPI implementation on these systems. However, as they acknowledge in their paper, these solutions fail for larger numbers of processes and for large message sizes.

1.1 Contributions

This thesis proposes that in order to efficiently develop (or port) a salable parallel application on a particular distributed memory architecture, it is necessary to have a thorough understanding of the behavior of the MPI primitives offered by its MPI implementation, available system level buffers, and other hardware constraints.

We investigate the behavior of non-blocking point to point communication primitives provided by two popular MPI implementations: the public domain MPICH [15]

implementation on a Linux cluster, and the proprietary IBM implementation on an IBM SP2 [10], and show how the same MPI calls offered by these two implementations differ in semantics. We then propose implementation specific usage strategies for these primitives than can reduce processor synchronization and optimize application performance, and use the proposed strategies to optimize the performance of scientific/engineering simulations using finite difference methods on structured adaptive meshes [1]. Finally, we propose a design for an independant multi-threaded communication engine for future architectures like IBM Blue Gene/L.

1.2 Organization

The thesis is organized as follows. Chapter 1 is the introduction. Chapter 2 explains the operation of generic non-blocking MPI message passing primitives. Chapter 3 investigates the performance characteristics of the MPICH implementation of non-blocking MPI calls on a Linux-based Beowulf cluster, and the IBM implementation of these calls on the IBM SP2 System. This section also presents usage strategies to optimize communication performance for each of these implementations. Chapter 4 evaluates the proposed strategies by applying them to optimize SAMR applications. Chapter 5 presents a design for a multi-threaded communication engine on architectures like IBM Blue gene/L which offer parallel hardware. Chapter 6 presents a summary of this thesis and comments on the future work.

Chapter 2
Message Passing in MPI

2.1 Overview of point to point communication

The elementary communication operation in MPI is point to point communication, that is, direct communication between two processors, one of which sends and the other receives. Point to point communication in MPI is ”two-sided”, meaning that both an explicit send and an explicit receive are required. In a generic send or receive, a *message* consisting of some block of data is transferred between processors. A message consists of an *envelope*, indicating the source and destination processors, and a *body*, containing the actual data to be sent. MPI uses three pieces of information to characterize the message body in a flexible way:

1. **Buffer*** - the starting location in memory where outgoing data is to be found (for a send) or incoming data is to be stored (for a receive).

2. **Datatype** - the type of data to be sent. In the simplest cases this is an elementary type such as float/REAL, int/INTEGER, etc. In more advanced applications this can be a user-defined type built from the basic types.

3. **Count** - the number of items of type datatype to be sent.

MPI specifies a variety of communication modes [14] that define the procedure used to transmit the message, as well as a set of criteria for determining when the communication event (i.e., a particular send or receive) is complete. For example, a *synchronous* send is defined to be complete when receipt of the message at its destination has been acknowledged. A *buffered* or *asynchronous* send, however, is complete when the outgoing data has been copied to a (local) buffer; nothing is implied about the arrival of the message at its destination. In all cases, completion

of a send implies that it is safe to overwrite the memory areas where the data were originally stored. The four communications modes available are:

- Standard
- Synchronous
- Buffered or Asynchronous
- Ready

In addition to the communication mode used, a send or receive may be *blocking* or *non-blocking*. A blocking send or receive does not return from the subroutine call until the operation has actually completed. Thus it ensures that the relevant completion criteria have been satisfied before the calling process is allowed to proceed. On the other hand, a non-blocking send or receive returns immediately, with no information about whether the completion criteria have been satisfied. This has the advantage that the processor is free to do other things while the communication proceeds "in the background".

The *standard* communication mode is MPI's general-purpose send mode. The other send modes, discussed above, are useful in special circumstances, but none have the general utility of standard mode. The vendor is free to implement the standard mode as asynchronous or synchronous depending on the message size and resource availability. Consequently, most parallel applications use the standard mode of communication. Also, all of these modes are mainly used to ensure correctness in a program; however, they can be treated the same if we base them on performance characteristics alone [7]. Thus, it turns out that the results obtained from modelling the standard mode sends/receives can be extended to other send/receive protocols as well.

2.2 Asynchronous non-blocking MPI communication

Use of asynchronous non-blocking MPI communication to overlap communication with computation is a primary strategy to improve parallel program performance.

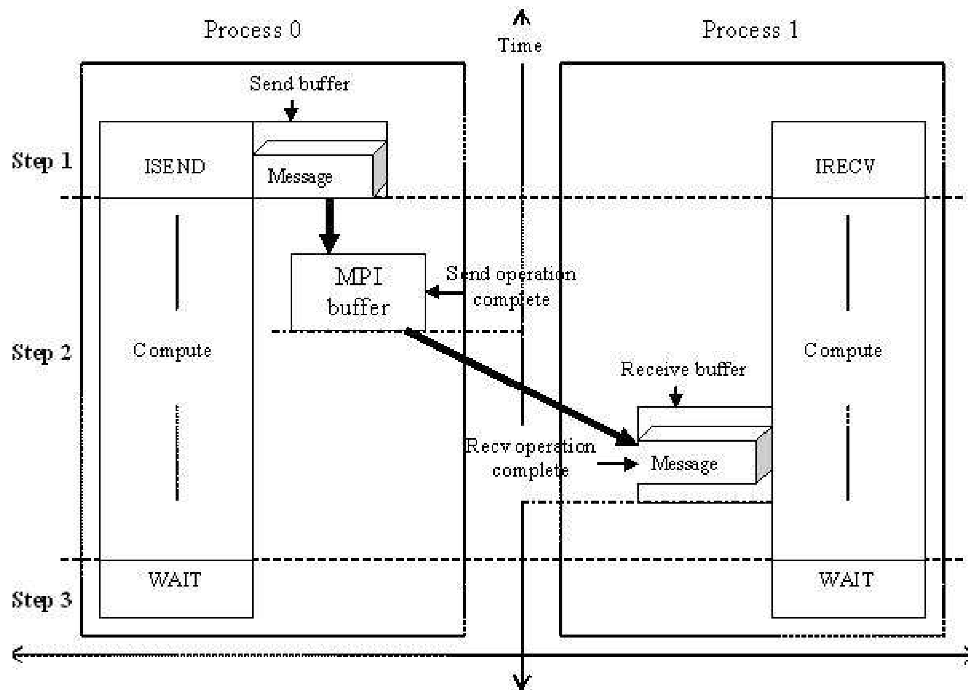

Figure 2.1: Schematic of a generic implementation of MPI non-blocking communications.

When MPI executes a standard mode send, one of two things happens. Either the message is copied into an MPI internal buffer and transferred asynchronously to the destination process (called asynchronous or buffered send) or both the source and destination processes synchronize on the message (called synchronous or unbuffered send). The MPI implementation is free to choose between buffering and synchronizing, depending on message size and resource availability. Most MPI vendors, however, use a scheme by which a standard send is asynchronous in nature up to a certain message size and then switches to synchronous mode for very long messages.

The generic operation of a non-blocking MPI communication is illustrated in Figure 2.1 as a "three step" process. This reflects the typical scenario as prescribed by parallel programming texts [16][5].

In the first step, process 0 (sender) posts a non-blocking send call (MPI_Isend)

which initiates the asynchronous send operation. Process 1 (receiver) posts a matching non-blocking receive (MPI_Irecv). Next, both processors can proceed with executions unrelated to the message being passed. Meanwhile, the communication subsystem delivers the message from process 0 to process 1. Finally, in step 3, both processors test (MPI_Test) or wait for (MPI_Wait) the completion of their respective send and receive operations.

As illustrated in Figure 2.1, the implementation of the non-blocking communication uses system and/or application buffers at the sender and the receiver. When the send is initiated, the message is copied into an internal MPI buffer at the sender. The communication subsystem then asynchronously transfers the message from the buffer at the sender to a buffer at the receiver. The MPI standard specifies that an asynchronous send operation completes as soon as the message is copied into the internal buffer, while the corresponding receive operation completes when the buffer allocated for the operation receives the data. This allows the sender and receiver operations to be decoupled, 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. However the exact thresholds and mechanisms of this synchronization are implementation dependent. As a result, fully exploiting the benefits of MPI's non-blocking communications requires an understanding of its implementations on the specific platform.

Chapter 3

Understanding the Behavior and Performance of Non-blocking MPI Communications

In this chapter we experimentally investigate the behavior and performance of non-blocking MPI communications provided by two popular MPI implementations: MPICH version 1.2.5, release date January 6, 2003 on a Beowulf cluster, and IBM MPI version 3 release 2, on the IBM SP2. The structure of the test kernel used for these experiments is illustrated in Figure 3.1.

3.1 Test Kernel

In this kernel the sending process (process 0) issues MPI_Isend (IS) at time-step T0 to initiate a non-blocking send operation. At the same time, 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. Finally we ran the experiments with both processes exchanging messages (each posts a send operation to the other). 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

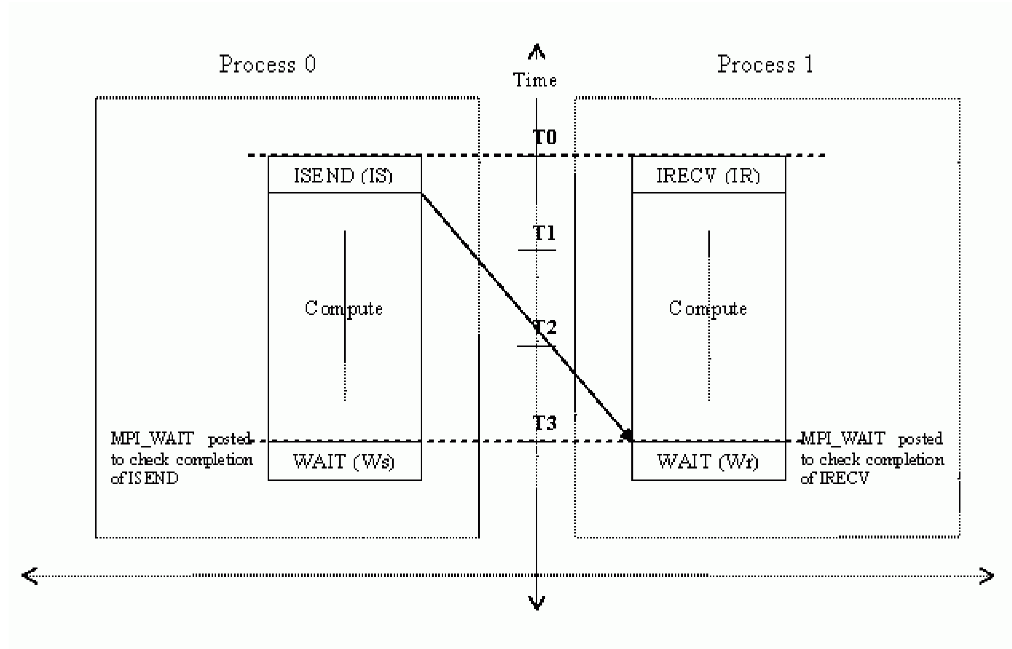

Figure 3.1: Operation of the test kernel used in the experimental investigation.

deadlocks due to synchronization. The results of these tests using the two MPI implementations are summarized and discussed in the rest of this section. Optimization strategies for each of these implementations are also presented.

3.2 MPICH on a Linux Beowulf Cluster

The first MPI implementation analyzed is MPICH version 1.2.5, release date January 6, 2003 [15] 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 nodes are connected using 100 MBPS switched TCP/IP interconnects. The MPICH profiling tool Upshot [8] is used for the profiles and timing graphs presented below.

3.2.1 Behavior of Non-Blocking Primitives

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

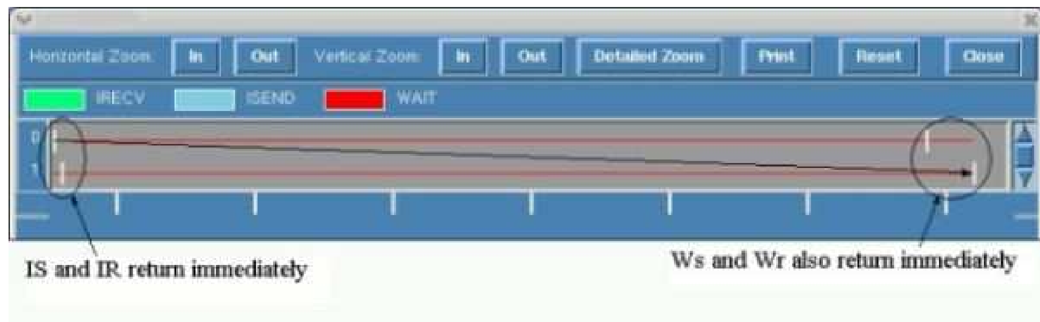

Figure 3.2: Profile of the test on MPICH where process 0 (top) sends a 1 KB message to process 1 (bottom).

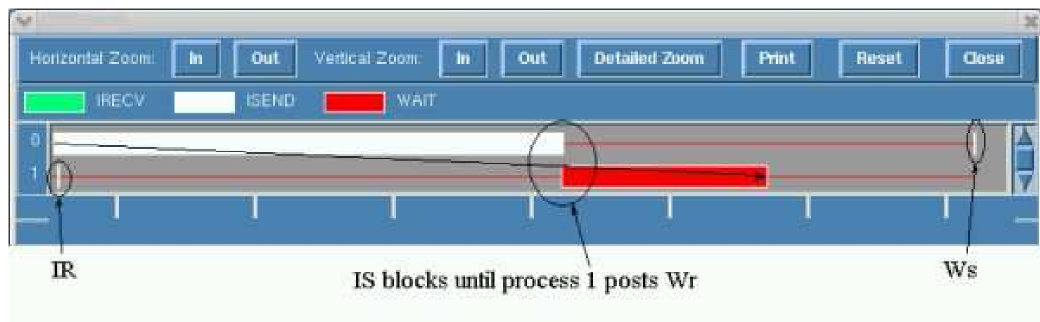

Figure 3.3: Profile of the test on MPICH where process 0 (top) sends a 60 KB message to process 1 (bottom).

observe the expected non-blocking semantics. As seen in the Upshot plots in Figure 3.2, IS and IR return without blocking. Furthermore, Ws and Wr, posted after local computations, return almost immediately, indicating that the message was delivered during the computation.

However, for message sizes greater than or equal to 60 KB, it is observed that IS blocks and returns only when the receiver process posts Wr. This behavior is illustrated in Figure 3.3. We can further see from the Figure that Wr blocks until the message delivery completes. Note that this threshold is dependent on the system buffer size as discussed below. We repeated the experiments using MPI buffered communication modes (MPI_Ibsend) with user defined buffers. The results demonstrated the same blocking behavior for larger messages as shown in Figure 3.4.

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 3.5, in this case MPI_Isend returns as soon as

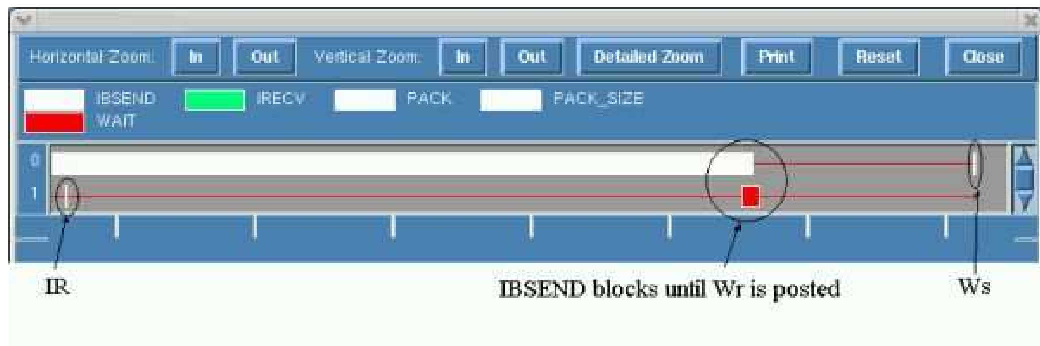

Figure 3.4: Profile of the test on MPICH in which buffered mode send is used.

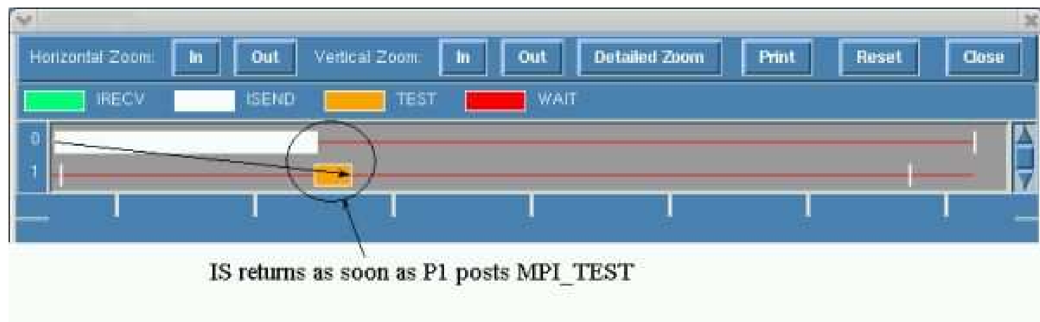

Figure 3.5: Profile of the test on MPICH in which process 1 (bottom) posts an intermediate MPI_Test

MPI_Test is posted. It was also seen that the Wr posted after computation 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 blocking behavior of non-blocking sends in the case of large message sizes can potentially lead to deadlocks. This may happen if two processes simultaneously send large messages to each other using MPI_Isend and block. The experiment plotted in Figure 3.6 evaluates this case. The Figure shows that process 1 initially blocks but

Figure 3.6: Profile of the test on MPICH in which both processes post MPI_Isend. Message size is 60Kb.

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 time out mechanism to ensure progress. However, which of the two processes will timeout is not deterministic.

The key factor affecting the message threshold that causes the non-blocking send to switch to blocking semantics is the size of the system socket buffer. 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¿. We repeated the experiments using messages of size 60KB but increasing the TCP socket buffer size. We observed that for a TCP socket buffer size of 64KB, IS did not block.

3.2.2 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 [6] consisting of (a) a high level MPI API layer (b) a middle Abstract Device Interface (ADI) layer and, (c) a Device layer. The Device layer provides a small set of core message passing routines, which are used by the higher layers to implement the MPI specification. The Frea Beowulf cluster uses the P4 device [2] as the Device layer, which 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. By the time MPI_Wait (Wr) is posted on the receiving processor, the network has delivered the message. As a result, both Ws and Wr take minimal time to return.

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. An inspection of the MPICH source code confirmed this behavior - when MPI_Isend cannot buffer the message using the system buffers, it switches to the P4 routine called by its synchronous counterpart MPI_Send, which blocks at the device layer waiting for an acknowledgement from the receiver. This is the reason why in Figure 3.3, MPI_Isend at process 0 blocks until MPI_Wait is posted on processor 1. The MPI_Wait completion time in this Figure is the time taken for message delivery. 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 64KB, 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.

3.2.3 Optimization Strategies for Non-blocking communications in MPICH

Message sizes in most real world applications are typically of order of hundreds of kilobytes. As a result, a naive use of non-blocking communications that does not take the blocking behavior of the MPI_Isend implementation into account can result in unexpected program behavior and a significant degradation of application performance.

```
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_*)
```

Figure 3.7: Example of a non-blocking implementation on MPICH

While the developer may stagger sends and receives to overlap computation with communication expecting the MPI_Isend to return immediately and provide asynchronous communication semantics, the send operations would block. This is especially true for applications that have irregular communication patterns (e.g. adaptive mesh applications) and some load imbalance. Based on the analysis presented above we identify two usage 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. In MPICH, a process uses a separate socket for every other process and therefore uses separate socket buffer for every other process. Hence, 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 3.7) for a typical loose-synchronous application, for example, a finite-difference PDE

```
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_*)
```

Figure 3.8: Optimization of the non-blocking implementation on MPICH

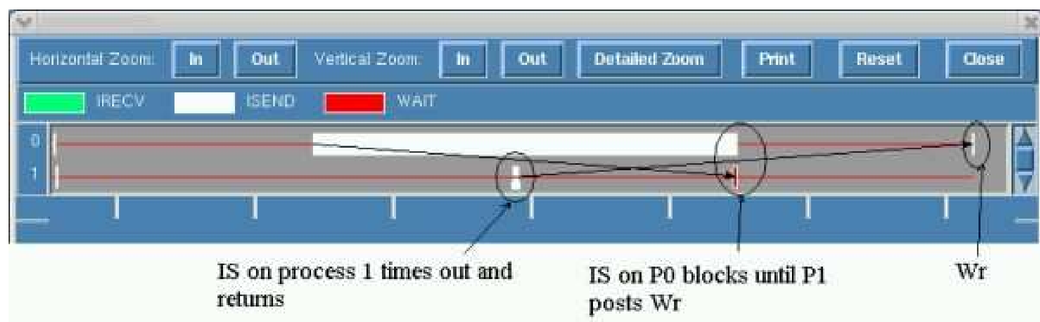

Figure 3.9: Example of a non-blocking implementation on MPICH

solver using ghost 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. This is done using the MPI_Isend/MPI_Wait pair. The process may do some further local computation and then finally waits to updates 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. This is shown in Figure 3.9. If we now insert an intermediate MPI_Testall call as shown in Figure 3.8, the MPI_Isend returns as soon as the receiver posts the test (see Figure 3.10). While the MPI_Testall call does have a cost, in our experience this cost is small compared to the performance gain.

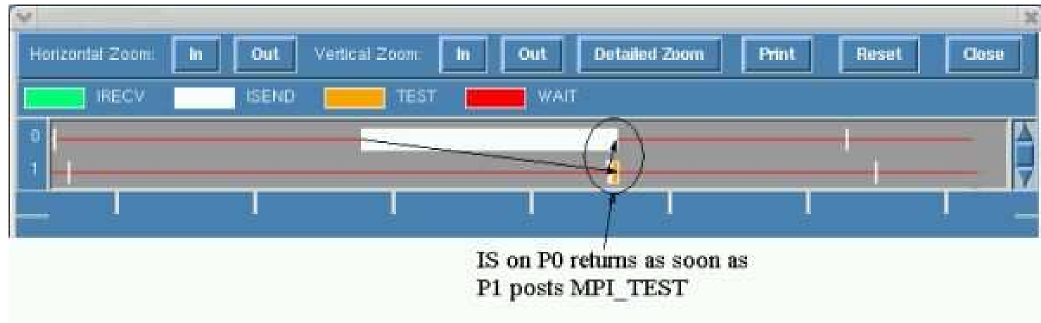

Figure 3.10: Optimization of the non-blocking implementation on MPICH.

3.3 The Parallel Operating Environment (POE) on IBM SP2

The second MPI implementation analyzed is the IBM native implementation (version 3 release 2) [10] on the IBM SP2, BlueHorizon, at the San Diego Supercomputing Center. Blue Horizon is a teraflop-scale Power3 based clustered SMP system at the San Diego Supercomputing Center. The machine consists of 1152 processors arranged as 144 8-way SMP compute nodes running AIX, each having 512 GB of main memory. The nodes are connected via a proprietary switched interconnect.

3.3.1 Behavior of Non-Blocking Primitives

Once again, our first experiment investigates the effect of message size on non-blocking communication semantics. As in the case of MPICH, 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. For smaller message sizes (1KB), we observe the expected non-blocking semantics. As seen in Figure 3.11, IS and IR return immediately. Furthermore, Ws and Wr posted after local computations also return almost immediately, indicating that the message was delivered during the computation. This behavior is also true for larger messages sizes (greater than or equal to 100 KB), i.e. the IBM MPI_Isend implementation continues to return immediately as per the MPI specification, and Ws and Wr take minimal time (of the order of microseconds) to return.

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

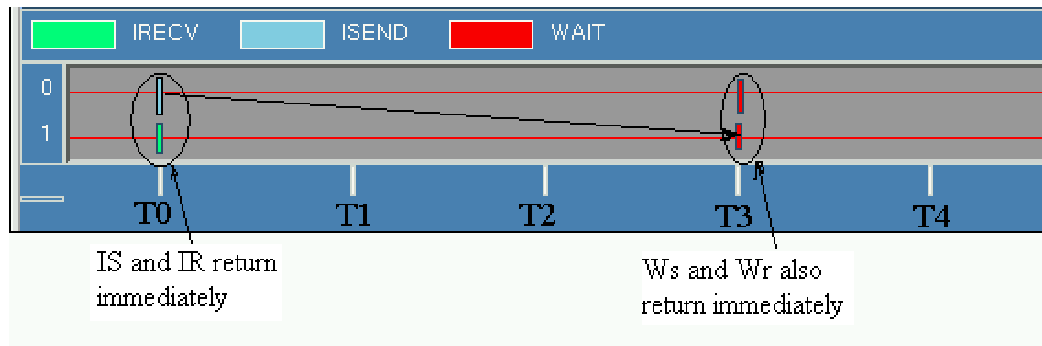

Figure 3.11: Profile of the test on SP2 where Ws and Wr are posted at the same time-step.

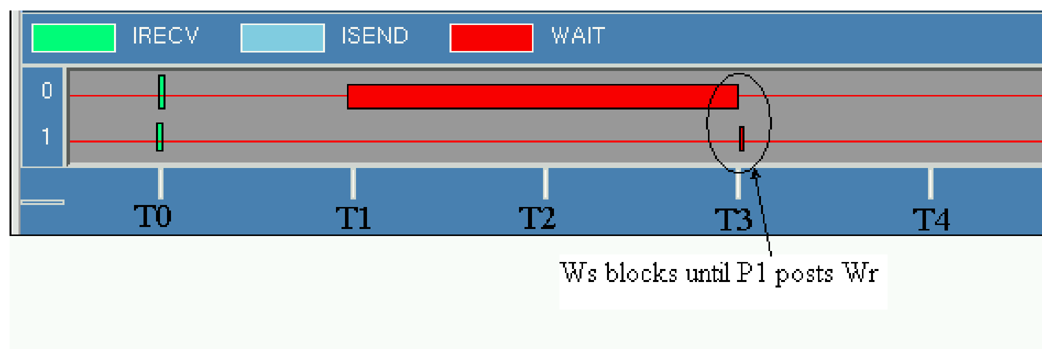

Figure 3.12: Profile of the test on SP2 where Ws and Wr are posted at different time-steps.

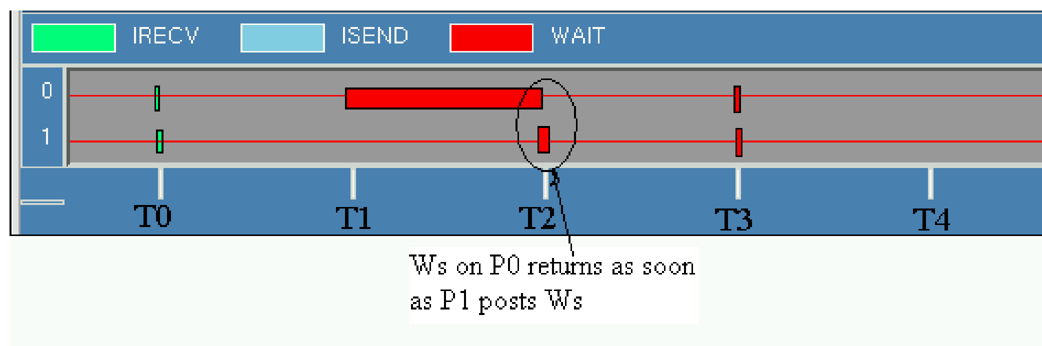

Figure 3.13: Profile of the test on SP2 showing the completion semantics of Ws.

the send to simulate the situation where one might want to reuse the send buffer. Wr remained at T3. In this case, the non-blocking behavior remained unchanged for small messages. However, for message sizes greater than or equal to 100KB, it was observed that Ws blocked until Wr was posted by the receiver at T3. This behavior is illustrated in Figure 3.12.

We now modify the experiment so that the two processes exchange messages. As shown in Figure 3.13, both processors post IS and IR at T0. Process 0 posts Ws at T1 while process 1 posts Ws at T2. Both processes then post Wr at T3. The message size is kept at 100KB. This test illustrates an interesting behavior. As seen in Figure 3.13, Ws posted at T1 on process 0 blocks. However, instead of waiting for Wr to be posted at T3, it returns as soon as process 1 posts Ws at T2.

3.3.2 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. This limit is directly dependant on the size of the memory that MPI uses and the number of processes in the execution environment [10]. For message sizes less than this eager limit, the messages are sent asynchronously. When message sizes exceed the eager 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 about, 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.

```
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_*)
```

Figure 3.14: Example of a non-blocking implementation on SP2.

3.3.3 Optimizations Strategies for Non-blocking Communications in the IBM POE

It is clear from the analysis above that the time spent waiting for the processes to synchronize, rather than network latency, is the major source of the communication overheads. Once again, this problem is particularly significant in applications where the communications are not completely synchronized (regular) and there is some load imbalance. The POE users' guide [9] specifies the environment variable, MP_EAGER_LIMIT, which defines the size of MPI messages that can be sent asynchronously. However, trying to increase this limit is not a scalable solution. This is because MP_EAGER_LIMIT is directly related to another environment variable, MP_BUFFER_MEM, which is the size of memory that MPI uses. 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 appropriating positioning IS, IR, Ws and Wr calls. The basic strategy consists of delaying Ws until after Wr and is illustrated in Figures 3.14 and 3.15. To illustrate the strategy, consider a scenario in which two processes use this template to exchange 3 messages each. Let IR1, IR2 and IR3 denote the MPI_Irecv calls, IS1, IS2 and IS3 denote the MPI_Isend

```
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_*)
```

Figure 3.15: Optimization of the non-blocking implementation on SP2.

calls and Ws1, Ws2, Ws3, Wr1, Wr2 and Wr3 denote the MPI_Wait calls. Wall denotes a MPI_Waitall call. As before, we split the execution sequence into steps T0 - T3.

Let both processes post IR1, IR2 and IR3 at T0. Assume that, due to load imbalance, process 0 computes until T2 while process 1 computes only until T1. Let the time taken by each MPI_Isend call to return be denoted by ts. As observed in the tests above, Ws1 posted on process 1 at T1+ts will block until process 0 posts Ws1 at T2+ts. Only after Ws1 returns can process 1 proceed with its send loop. Let the time taken by each of Ws2, Ws3 and Wall be denoted by tw and the cumulative message passing latency be μ. The equation for the time taken for process 1 to complete execution is thus T2+2*(ts+tw)+tw+μ. For large numbers of messages, this value can become significant. Consequently, to minimize the blocking overhead due to Ws1 on process 1 we must move it as close to T2 as possible.

If we now remove Ws from the send loop and post a collective MPI_Waitall as shown in Figure 3.15, we observe that since Ws is moved out of the loop, process 1 goes ahead and posts all of its non-blocking sends. Also, by the time its reaches T2, it has already posted IS0, IS1 and IS2 and is waiting on Wall (assuming that the time interval T2-T1 is sufficient to post all the non-blocking sends, followed by Wall). The time taken by process 1 to finish execution in this case is just T2+μ.

Chapter 4

Evaluation of communication performance in SAMR Applications

This chapter evaluates the optimization strategies proposed in chapter 3 by applying them to the the Grid Adaptive Computational Engine (GrACE) [18][17] framework, a toolkit for developing parallel SAMR applications. The rest of this chapter is organized as follows: Section 4.1 provides an overview of Structured Adaptive Mesh Refinement techniques for solving partial differential equations. Section 4.2 enumerates typical communication requirements in parallel SAMR applications. In section 4.3 we briefly describe the GrACE framework. Finally, in section 4.4, we integrate our optimization strategies within the GrACE framework and evaluate performance improvements using the RM3D SAMR application kernel [3].

4.1 SAMR Overview

Dr. Marsha Berger developed a formulation of the adaptive mesh refinement strategy for structured meshes [1] based on the notion of multiple, independently solvable grids, all of which were of identical type, but of different size and shape. The underlying premise of the strategy is that all the grids for any resolution that cover a problem domain are equivalent in the sense that given proper boundary information, they can be solved independently by identical means.

Dynamic Structured Adaptive Mesh Refinement (SAMR) techniques [1] for solving partial differential equations (PDE) provide a means for concentrating computational effort to appropriate regions in the computational domain. The numerical solution to a PDE is obtained by discretizing the problem domain and computing an approximate solution to the PDE at the discrete points. These methods (based on finite differences)

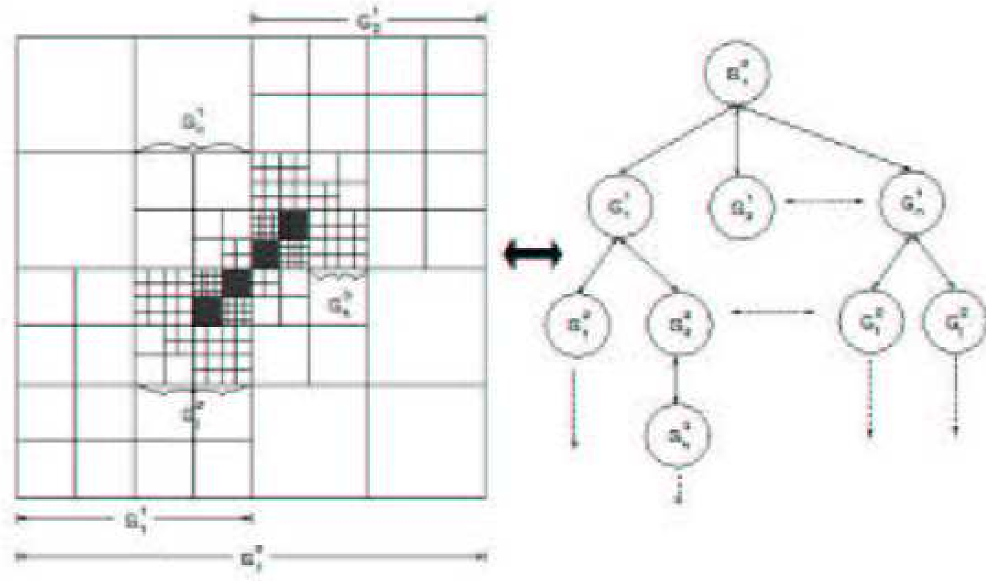

Figure 4.1: Grid Hierarchy.

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 tagged and finer grids are laid over these tagged regions of the coarse grid. Refinement proceeds recursively so that regions on the finer grid requiring more resolution are similarly tagged and even finer grids are overlaid on these regions. The resulting grid structure is a dynamic adaptive grid hierarchy as shown in Figure 4.1.

Berger's AMR scheme employs the nested hierarchy of the grids to cover the appropriate sub-domain at each level by the appropriate time step, then recursively advancing the next finer level by the appropriate time step, then recursively advancing the next finer level by enough iterations at its (smaller) time step to reach the same physical time as that of the newest solution of the current level. That is, the integrations at each level are recursively interleaved between the iterations at coarser levels. Thus, the Berger AMR approach refines in space and if the refinement factor between the finer level (l+1) and the next coarser level is r, then grids on the finer level (l+1) will be advanced r time steps for every coarser time step. For a d dimensional domain, the grids at level (l+1) must cover the same portion of the computational domain as only $1/r^d$ coarser cells at level l. For example, using a refinement factor of

2 on a three dimensional domain, 2 iterations at level 1 will take more computation time that an iteration at the root level (which comprises the entire computational domain) unless the grids at level 1 cover no more that 1/8 of the domain.

4.2 Communication Requirements of Parallel SAMR Applications

In an SAMR formulation, the grid hierarchy is refined both in space and in time. Refinements in the space create finer level grids which have more grid points/cells than their parents. Refinement in time means that finer grids take smaller time steps and hence have to be advanced more often. As a result, finer grids not only have greater computational loads but also have to be integrated and synchronized more often. This results in space and time heterogeneity in the SAMR adaptive grid hierarchy. furthermore, regridding occurs at regular intervals at each level and results in refined regions being created, moved and deleted. Together, these characteristics of SAMR applications makes their efficient implementation a significant challenge.

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 [13][18]. Each processor starts at the coarsest level, integrates the patches at this level and performs intra-level or ghost communications to update the boundaries of the patches. It then recursively operates on the finer grids using the refined time steps - i.e. for each step on a parent grid, there are multiple steps (equal to the time refinement factor) on the child grid. when the parent and child grid are at the same physical time, inter-level communications are used to inject the information from the child to its parent. Dynamic re-partitioning and re-distribution is typically required after this step.

The overall performance of parallel SAMR applications is limited by the ability to partition the underlying grid hierarchies at runtime to expose all inherent parallelism, minimize communication and synchronization overheads, and balance load. A critical requirement of the load partitioner is to maintain logical locality across the

partitions at different levels of hierarchy and the same level when they are decomposed and mapped across processors. The maintenance of locality minimizes the total communication and synchronization overheads.

4.2.1 Communication overheads in parallel SAMR applications

The communication overheads of parallel SAMR applications primarily consist of 3 components:

1. *Inter-level communications* defined between component grids at different levels of the grid hierarchy and consist of prolongations (coarse to fine transfer and interpolation) and restrictions (fine to coarse transfer and interpolation).

2. *Intra-level communications* required to update the grid elements along the boundaries of local portions of a distributed grid, consists of near-neighbor exchanges. These communications can be scheduled so as to be overlapped with computations on the interior region.

3. *Synchronization cost*, which occurs when the load is not well balanced among all processors. These costs may occur at any time step and at any refinement level due to the hierarchical refinement of space and time in SAMR applications.

Clearly, an optimal partitioning of the SAMR grid hierarchy and scalable implementations of SAMR applications requires careful consideration of the synchronization costs incurred while performing intra and inter level communications.

4.3 Grid Adaptive Computational Engine

The Grid Adaptive Computational Engine (GrACE) infrastructure at Rutgers University is an approach to distributing AMR grid hierarchies. GrACE is an object oriented tool kit for the development of parallel SAMR applications. It is built on a "semantically specialized" distributed shared memory substrate that implements a hierarchical distributed dynamic array (HDDA) [18].

```
{
 /*Initiate all non-blocking receives before hand*/
 LOOP
 MPI_Irecv();
 END LOOP
 for each data operation {
   for all neighboring processors {
      Pack local boundary values
      /*Send boundary values*/
      MPI_Isend();
      /*Wait for completion of my sends*/
      MPI_wait();
      Free the corresponding send buffer
   }
   /*Wait for completion of all my receives*/
   MPI_Waitall();
   Unpack received data
 }
 *****COMPUTATION CONTINUES*****
}
```

Figure 4.2: Intra-level Communication model in a parallel SAMR application framework.

Due to irregular load distributions and communication requirements across levels of the grid hierarchy in parallel SAMR applications, GrACE makes extensive use of non-blocking MPI primitives to reduce synchronization overheads. For example, intra-level communications can be scheduled so as to be overlapped with computations on the interior region.

4.3.1 Intra-level communication model

A typical model of the intra-level communication implemented by GrACE is illustrated in Figure 4.2.

In this implementation, each processor maintains local lists of all messages it has to send and receive to/from its neighboring processors. As seen in Figure 4.2, each process first posts non-blocking receives (MPI_Irecv). It then goes through its send list, packs each message into a buffer and sends it using MPI_Isend. Typical message sizes in these applications are in the order of hundreds of Kilobytes. Following each MPI_Isend, a corresponding MPI_Wait is posted to ensure completion of the send operation so that the corresponding send buffer can be freed. Once all the sends are

```
{
/*initiate all non-blocking receives
 before hand*/
LOOP
MPI_Irecv();
END LOOP
for each data operation{
 for all neighboring processors{
   Pack local bounary values

   /****INTERLEAVE NON-BLOCKING TESTS
    FOR COMPLETETION OF RECEIVES*****/
    MPI_Test();

   /*Send local boundary values*/
    MPI_Isend();

   /*Wait for completion of my sends*/
    MPI_Wait();
    Free the corresponding send buffer
   }
   /*Wait for completion of all my receives*/
    MPI_Waitall();
    Unpack received data
  }
 *****COMPUTATION CONTINUES*****
}
```

Figure 4.3: Staggered Sends - Optimizing SAMR Intra-level Communication for MPICH.

completed, a MPI_Waitall is then posted to check completions of the receives. This exchange is a typical ghost communication associated with parallel finite difference PDE solvers as described in Chapter 3. Clearly, the optimizations proposed by us Chapter 3 can be effective applied here to reduce the synchronization costs.

4.3.2 Optimizations

Figures 4.3 and 4.4 illustrate optimizations to the intra-level message passing algorithm in GrACE for Frea and Blue Horizon.

Staggered Sends on Frea - As discussed in Section 3.2.3, an efficient strategy - on MPICH - to release a blocked MPI_Isend is for the receive side to post intermediate non-blocking calls to MPI_Test. Figure 4.3 shows how the SAMR algorithm is

```
{
/*initiate all non-blocking receives
 before hand*/
LOOP
MPI_Irecv();
END LOOP
for each data operation{
 for all neighboring processors{
   Pack local bounary values
   /*Send local boundary values*/
    MPI_Isend();

   /***The wait call to check completion
    is moved out of the loop****/
   }
   /*Wait for completion of all my receives*/
    MPI_Waitall();
    Unpack received data

  /*DELAYED WAITS - Check completion of sends*/
    MPI_Waitall();
    Free send buffer;
  }
 *****COMPUTATION CONTINUES*****
 }
```

Figure 4.4: Delayed Waits - Optimizing SAMR Intra-level Communication for IBM POE.

optimized by interleaving an MPI_Test with each MPI_Isend that the ghost iterator posts. (For further explanation refer 3.2.3)

Delayed Waits on Blue Horizon - We delayed the MPI_Wait call (that checks completion of each MPI_Isend), originally inside the ghost iterator, to until after the receives are completed (Figure 4.4). As discussed in Section 3.3.3, this allows the sending side to post a larger number of MPI_Isends before being forced to synchronize.

Similar optimizations have been incorporated within the inter-level message passing algorithm. The next section discusses improvements in efficiency obtained by these architecture specific optimizations.

4.4 Evaluation using the RM3D SAMR 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 techniques to solve the Richtmyer-Meshkov instability [3]. 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 Chapter 3, on both Frea and Blue Horizon. Except for the optimizations in the intra and inter level communication algorithms in GrACE, all other application-specific and refinement-specific parameters are kept constant. Both evaluations use 3 levels of factor 2 space-time refinements with regriding performed every 8 time-steps at each level. The results of the evaluation for MPICH on Frea for 16, 32 and 64 processors are shown in Figure 4.5. The figure compares the original execution times and communication times of the application with the ones obtained by optimizing the message passing algorithm. These runs used a base grid size of 128*32*32 and executed 100 iterations. We observe that the reduction in communication time is 27.32% on an average.

On the SP2 the evaluation run used a base grid size of 256*64*64 and executed 100 iterations. Figure 4.6 shows the comparisons of the execution times and communication times respectively for 64, 128 and 256 processors. In this case we observe

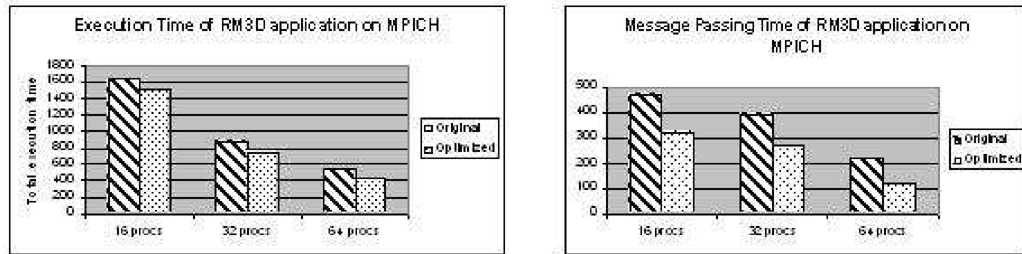

Figure 4.5: Comparison of execution and communication times on Frea (MPICH)

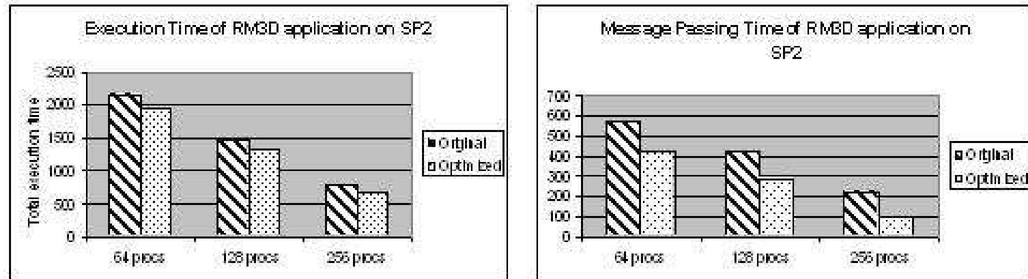

Figure 4.6: Comparison of execution and communication times on SP2 (IBM POE)

that the reduction in communication time is 44.37% on the average.

Furthermore, it can be clearly seen from both the figures that the decrease in execution time is primarily due to the decrease in message passing time.

Chapter 5

Future work - Design of an independent multi-threaded communication engine for parallel hardware architectures

5.1 Introduction

AMR grids are inherently heterogeneous, varying in both resolution and extent and can be moved, created and deleted on fly. SAMR applications offer multiple levels of granularity and parallelism. Grids at the same level of refinement can be operated in parallel. Similarly composite slices across all refinement levels (i.e. a parent grid and all its children) can also be operated on in parallel. Finally, each grid can itself be operated on in a data-parallel fashion. The SAMR algorithm requires that each grid be periodically synchronized with its parents and its neighboring siblings requiring communication at regular intervals. Clearly, there is a need for a runtime communication engine that can exploit these many levels and granularities of parallelism, and efficiently manage and overlap the synchronizations and communications with computations.

Multi-threading is an approach which can best exploit the parallelism inherent in SAMR applications. The advantages of using threads are discussed in detail in a number of publications [21][12]. Among the obvious are the easy use of multiple processors if available, latency hiding and cheap inter-thread (as opposed to inter-process) communication and synchronization. In addition to this, the MPI specification is "highly parallelizable" itself, in the sense that communication operations are independent of each other. This makes it natural to try performing computation and communication in parallel, using all the concurrency that hardware can provide.

In order to design a reliable multi-threading algorithm it is necessary that the

MPI implementation be thread safe. The IBM POE offers thread safe libraries for its MPI calls while there have been a number of papers published towards designing a thread-safe version of MPICH [21][12][19]. However, a major restricting factor while using multi-threading to exploit inherent parallelism in current architectures like the Beowulf and IBM SP2 discussed in Chapter 3 is that a single processor is used to carry out both communication and computation. This leads to a single process executing user application as well as the MPI communication subsystem. Consequently, use of multi-threading to exploit inherent parallelism in applications does not yield major performance improvement on these architectures, even if the MPI implementation is thread safe.

Future projects like the IBM Blue Gene/L supercomputer [20] being built at the Lawrence Livermore Laboratory identify this performance bottleneck and are working towards providing an architecture design in which each chip includes two processors: one for computation and other for communication. This chapter presents a design of a multi-threaded communication engine that can be plugged into the GrACE framework to take advantage of the concurrent hardware provided by future parallel architectures like Blue Gene/L.

5.2 Algorithm

The prototype of our algorithm has been implemented for intra-level ghost communications in SAMR applications using the POSIX threads library, and has been tested for reliability on Linux. It has been designed such that it can be directly plugged into the GrACE framework as a separate communication engine. Figure 5.1 shows a schematic of the prototype.

Immediately after MPI_Init is posted, the main thread spawns a separate thread (called the communication thread) which handles all the MPI calls. Both threads share a common data-structure called the Sync queue. The main thread proceeds with the sequential program while the communication thread constantly polls the Sync queue for work. When the main thread encounters a ghost synchronization call, it enqueues the boundary parameters in the sync queue and immediately continues with

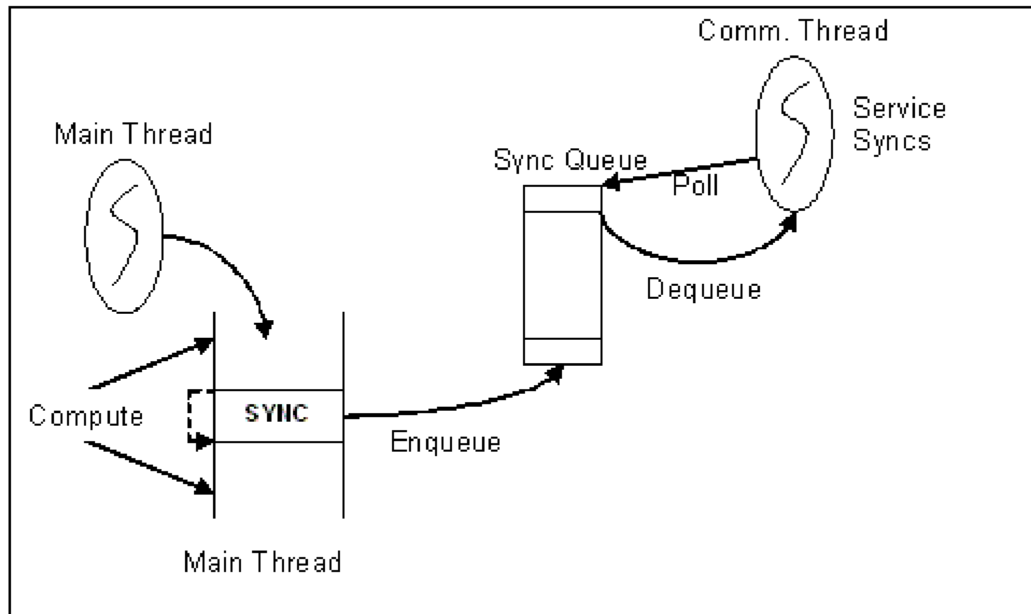

Figure 5.1: Schematic of a multi-threaded algorithm for SAMR applications

the the next program instruction (computation). Concurrently, the communication thread dequeues the boundary parameters and executes the ghost iterator described in Section 4.3.1 to carry out the boundary exchange. This algorithm exploits the fact that composite slices of grids across different levels of refinement can be operated upon in parallel. Hence, computation at a higher level can proceed concurrently with ghost communication at a lower level.

Access to the Sync queue can be synchronized using a mutex lock, a semaphore or a condition variable. However, while implementing a semaphore or a condition variable, care should be taken to avoid conflict between the user interrupts used by the MPI communication subsystem and the semaphore or condition variable implementation. For example, we had to configure MPICH to use the SIGUSR2 interrupt for its socket calls in order to avoid a conflict with the gnu semaphore implementation which uses SIGUSR1 to wake up a thread.

In order to test our prototype for correctness on the current non thread-safe MPICH implementation on Frea, we used only one communication thread which would post all the MPI calls serially. However, parallelism in SAMR applications can be further exploited on a thread safe version of MPI by implementing two different queues, one for the sends and another for the receives. Thus multiple communication

threads could poll these queues for work and sends and receives can be carried out simultaneously achieving greater overlap.

5.3 Future Work

The research presented in this chapter is but a small step in the field of using multi-threading for performance optimization. The future direction would be to port the algorithm on an architecture which offers concurrent hardware and implements a thread safe MPI library. At the time of this writing the authors are corresponding with IBM Research for access to the IBM Blue Gene/L simulator in order to evaluate the current prototype and implement a full fledged engine which would exploit all ingrained parallelism using multi-threading.

Chapter 6
Conclusions

The message passing programming model is the widely accepted paradigm for developing high performance parallel and distributed applications. Furthermore, the Message Passing Interface (MPI) has evolved as the de-facto message passing standard for supporting portable parallel applications. Commercial as well as public-domain implementations of the MPI specification are available for most existing parallel platforms including clusters of networked workstations and high-performance systems. However, 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. MPI's non blocking communication operations are often used by applications with irregular loads and/or communications as they allow the application to reduce synchronization overheads and hide latency. However, these operations are particularly sensitive to the implementation details. As a result, naive use of these operations without an understanding of the underlying implementation can result in serious performance degradations.

Current clusters like Beowulf and SP2 use the same hardware for computation as well as communication. As a result, there is a single process executing the user application as well as the MPI communication subsystem. However, future architectures like IBM Blue Gene/L plan to offer parallel hardware for computation and communication. The obvious approach to reduce communication synchronization on these clusters would be to use separate threads for computation and communication.

In this paper we experimentally analyzed the behavior and performance of non-blocking communication provided by two popular MPI implementations: the public domain MPICH [15] implementation on a Linux cluster, and the proprietary IBM

implementation on an IBM SP2 [10]. We then proposed and evaluated usage strategies for these primitives than can reduce processor synchronization and optimize application performance using these implementations of MPI. We used the proposed strategies 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 27.32% for MPICH and 44.37% for IBM MPI. Finally, we proposed design for a multi-threaded communication engine that can exploit ingrained parallelism in SAMR applications on architectures that offer separate hardware for communication and computation.

References

[1] M. Berger and J. Oliger. Adaptive Mesh Refinement for Hyperbolic partial Differential Equations. *Journal of Computational Physics*, 53:484–512, 1984.

[2] Installation and User's Guide to MPICH, a Portable Implementation of MPI Version 1.2.4 - the chp4 device for Workstation Networks. , 2001. http://www-unix.mcs.anl.gov/mpi/mpich/docs.html.

[3] 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(1):39–50, Jan 2002.

[4] W.K. Giloi. Parallel supercomputer architectures and their programming models. Parallel Computing, 20(10-11):1443-1470, 1994.

[5] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI Portable Parallel Programming with the Message-Passing Interface.* The MIT Press, Cambridge, Massachusetts, London, England, 1999.

[6] W. Gropp, E. Lusk, A. Skjellum, and N. Doss. MPICH: A High-Performance, Portable Implementation for MPI Message-Passing Interface. *Parallel Computing*, 22:789–828, 1996.

[7] D. Grove. Performance modelling of message-passing parallel programs, January 2003. Dissertation.

[8] V. Herrarte and E. Lusk. Studying Parallel Program Behavior with upshot. *ANL-91/15, Argonne National Laboratory*, 1991.

[9] IBM. Parallel Environment (PE) for AIX V3R2.0: MPI Programming Guide. Second Edition. , December 2001. POE.

[10] IBM. Parallel environment (pe) for AIX V3R2.0: Operation and Use, Vol. 1. , December 2001.

[11] J.B. White III and S.W. Bova. Where's the Overlap? An Analysis of Popular MPI Implementations. In *Proceedings of the Message Passing Interface Developer's and User's Conference Journal of Papers and presentations*, Atlanta, GA, March 1999.

[12] B. Lewis and D. J. Berg. Threads primer: A Guide to Multithreaded Programming. *Sunsoft Press*, 1996.

[13] X. Li and M. Parashar. Dynamic Load Partitioning Strategies for Managing Data of Space and Time Heterogeneity in Parallel SAMR Applications. In *Proceedings of 9th International Euro-Par Conference (Euro-Par 2003), Lecture Notes in Computer Science*, pages 181 – 188, Klagenfurt, Austria, August 2003.

[14] The MPI Forum. The MPI Message-Passing Interface Standard 2.0. , September 2001. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[15] MPICH - A Portable MPI Implementation. , Januray 2003. http://www-unix.mcs.anl.gov/mpi/mpich/.

[16] P. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[17] M. Parashar. GrACE - A Grid Adaptive Computation Engine. . http://www.caip.rutgers.edu/TASSL/Projects/GrACE.

[18] M. Parashar and J. C. Browne. Systems Engineering for High Performance Computing Software: The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh Refinement. In *IMA Volume 117: Structured Adaptive Mesh Refinement (SAMR) Grid Methods*, pages 1 – 18, January 2000.

[19] B. V. Protopopov and A. Skjellum. A multi-threaded Message Passing Interface (MPI) architecture: performance and program issues. In *Journal of Parallel and Distributed Computing*, pages 61(4): 449–466, 2001.

[20] IBM Research. IBM Blue Gene/L. http://www.research.ibm.com/bluegene.

[21] A. Skjellum, B. Protopopov, and S. Hebert. A thread taxonomy for MPI. In *Proceedings of MPI Developer's Conference*, pages 50–57, June 1996.