ONE-SIDED COMMUNICATION FOR HIGH PERFORMANCE COMPUTING APPLICATIONS

Brian W. Barrett

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University
March 2009

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Andrew Lumsdaine, Ph.D.

Randall Bramley, Ph.D.

Beth Plale, Ph.D.

Amr Sabry, Ph.D.

March 30, 2009

To my parents, who never let me lose sight of my goals.

Acknowledgements

I thank my parents, Daniel and Elizabeth Barrett, and sisters Colleen and Mary Ann, for their love and support, even when I did things the hard way.

I thank my colleagues in the Open Systems Laboratory at Indiana University and Sandia National Laboratories. In particular, Jeff Squyres, Doug Gregory, Ron Brightwell, Scott Hemmert, Rich Murphy, and Keith Underwood have been instrumental in refining my thoughts on communication interfaces and implementation for HPC applications.

I thank the members of my committee, Randy Bramley, Beth Plale, and Amr Sabry. They have provided extraordinary guidance and assistance throughout my graduate career.

Finally, I thank my advisor Andrew Lumsdaine for many years of mentoring, encouragement, and, occasionally, prodding. His support has been unwavering, even when my own faith was temporarily lost. This work would never have come to fruition without his excellent guidance.

This work was supported by a grant from the Lilly Endowment, National Science Foundation grants EIA-0202048 and ANI-0330620, Los Alamos National Laboratory, and Sandia National Laboratories. Work was also funded by the Department of Energy High-Performance Computer Science Fellowship (DOE HPCSF). Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Abstract

Parallel programming presents a number of critical challenges to application developers. Traditionally, message passing, in which a process explicitly sends data and another explicitly receives the data, has been used to program parallel applications. With the recent growth in multi-core processors, the level of parallelism necessary for next generation machines is cause for concern in the message passing community. The one-sided programming paradigm, in which only one of the two processes involved in communication actively participates in message transfer, has seen increased interest as a potential replacement for message passing.

One-sided communication does not carry the heavy per-message overhead associated with modern message passing libraries. The paradigm offers lower synchronization costs and advanced data manipulation techniques such as remote atomic arithmetic and synchronization operations. These combine to present an appealing interface for applications with random communication patterns, which traditionally present message passing implementations with difficulties.

This thesis presents a taxonomy of both the one-sided paradigm and of applications which are ideal for the one-sided interface. Three case studies, based on real-world applications, are used to motivate both taxonomies and verify the applicability of the MPI one-sided communication and Cray SHMEM one-sided interfaces to real-world problems. While our results show a number of short-comings with existing implementations, they also suggest that a number of applications could benefit from the one-sided paradigm. Finally, an implementation of the MPI one-sided interface within Open MPI is presented, which provides a number of unique performance features necessary for efficient use of the one-sided programming paradigm.

Contents

CHAPTER 1

Introduction

High performance computing (HPC), the segment of computer science focused on solving large, complex scientific problems, has long relied on parallel programming techniques to achieve high application performance. Following the growth of Massively Parallel Processor (MPP) machines in the late 1980s, HPC has been dominated by distributed memory architectures, in which the application developer is responsible for finding and exploiting parallelism in the application. The Message Passing Interface (MPI) has been the most common infrastructure used to implement parallel applications since its inception in the mid-1990s. [**31, 37, 61, 84**]

Recent changes in the HPC application space, basic processor design, and in MPP architectures have renewed interest in programming paradigms outside of message passing. [**5, 59**] Many in the HPC community believe MPI may not be sufficient for upcoming HPC platforms due to matching cost, synchronization overhead, and memory usage issues. A number of radically different solutions, from new communication libraries, to new programming models, to changes in the MPP architecture, have been proposed as viable alternatives to message passing as machines evolve.

Presently, parallel application developers are generally limited to MPI on large scale machines, as other interfaces are either not available or not well unsupported. The growth in potential programming options resulting from recent trends will produce more interface and paradigm choices for the application programmer. Such a wide range of options motivates the need to categorize both available programming paradigms and their suitability to particular classes of applications. This thesis begins that work, for a particular segment of the paradigm space, one-sided communication.

The remainder of this chapter examines the relative stability which has existed in HPC since the early 1990s (Section 1) and the forces driving the current uncertainty in the field (Section 2). The one-sided communication paradigm is briefly introduced in Section 3, and will be discussed in detail in Chapter 2. Finally, Section 4 provides an overview of this thesis as well as its contributions to the field.

1. Message Passing Reigns

In the mid 1980s and early 1990s, a number of companies, including nCUBE [**63**], Intel [**46**], Meiko [**60**], Thinking Machines [**42**], and Kendall Square Research [**74**], began marketing machines which connected a (potentially large) number of high speed serial processors to achieve high overall performance. These machines, frequently referred to as Massively Parallel Processor (MPP) machines, began to overtake vector machines in application performance. The individual processors generally did not share memory with other processors, and the programmer was forced to explicitly handle data movement tasks between processors.

Although more difficult to program than the auto-vectorizing Fortran of previous machines, the message passing paradigm which developed proved quite successful. The success of the model can largely be traced to its natural fit with HPC applications of the time. Applications were largely physics based, with static partitioning of physical space. At each time step, nearest neighbors exchanged information about the borders of the physical space, using explicit send/receive operations. However, each machine provided a different flavor of message passing, which made portable application development difficult. Application writers frequently had to change their code for every new machine.

The success of the message passing model led to the creation of the Message Passing Interface in 1994, eliminating much of the portability problem with distributed memory programming. MPI's ubiquity meant that application developers could develop an application on one platform and it would likely run with similar performance on other machines of the same generation, as well as the next generation of machines. The combination of a

natural fit to applications and the ubiquity of the message passing interface led to a large application base, all with similar communication requirements.

Likewise, MPI's ubiquity led system architects to design platforms which were optimized to message passing. Because message passing does not necessarily require a tightly coupled processor and network, system architects were able to leverage commodity processors coupled with specially designed interconnect networks. The Top 500 fastest supercomputers in Winter 2008 includes one machine which uses vector processors[1], while the remainder used commodity processors and message passing-based networks, showing the prevalence of the MPP model.

2. Growing Uncertainty

The HPC community has seen a long period of stability in machine architecture and programming paradigm, which has benefited both application developers and computer architects. Application developers have been able to concentrate on adding new simulation capability and optimizing overall performance, rather than porting to the next machine with its new programming model. Likewise, system architects were able to optimize the architecture for the relatively stable application workload.

Current trends in both system architecture and application workload, however, are disrupting the stability. New application areas are being explored for use with HPC platforms, including graph-based informatics applications, which require radically different programming models and network performance than traditional HPC applications. At the same time, processor architectures have changed to provide greater per-processor performance by providing more computational cores per processor, rather than through faster clock rates and serial performance. These multi-core processors shift the burden of increased per-processor performance to the programmer, who must now exploit both inter- and intra-processor parallelism.

[1]The Earth Simulator, which was the fastest machine for much of the 2000s, is the lone vector machine. While utilizing vector processors, it also provided distributed memory and a custom high-speed network between individual machines.

Multi-core processors in HPC are generally programmed by viewing them as a number of individual, complete processors. Message passing is utilized for communication, whether between nodes, processors within a node, or cores within a processor. Initial work suggests that there is a performance penalty for this model, but not significant enough to change current programs. [**40**] This is due, in part, to optimizations within the MPI libraries to exploit the memory hierarchy available in multi-core processors. [**23, 57**] Future processors, however, are likely to see the number of cores grow faster than both the memory bandwidth and outstanding memory operation slots, fueling the debate about programming future multi-core processors.

At the same time, the graph-based informatics applications are becoming more important within the HPC community and have a radically different communication model than more traditional physics applications. Traditional physics applications exchange messages at set points in the algorithm, generally at the conclusion of an algorithm's iteration, at which point the data which borders a processor's block of data must be shared with its neighbors. Informatics applications, however, frequently must communicate based on the structure of the graph, and a processor may need to talk to every other processor in the system during a single iteration. In addition, informatics applications generally send many more messages of a smaller size than do physics applications.

The growing concern over the programming paradigm used in multi-core designs, particularly as the per-core memory and network bandwidth shrinks with growing core count, has led many in the HPC field to suggest message passing many no longer be appropriate. Alternatives such as implicitly parallel languages [**21, 52**], hybrid message passing/threaded models [**19**], and alternative communication paradigms [**53, 58**] have all been proposed as solutions to growing performance problems.

3. One-Sided Communication

The one-sided communication paradigm is one of the alternative solutions to uncertainty in the HPC community. Message passing requires both the sender and receiver to be involved in communication: the sender describes the data to be sent as well as the target of

the message, and the receiver specifies the location in memory in which received data will be delivered. One-sided communication, however, requires only one of the two processes actively participate in the communication. The process initiating the one-sided transfer specifies all information that both the sender and receiver would specify with message passing. The target side of the operation is not directly involved in the communication.[2]

One-sided communication is seen as a potential solution to the multi-core issue because it reduces synchronization, discourages the use of bounce buffers which later require memory copies, and may be a better match to emerging informatics applications. One-sided communication implementations also have a performance advantage over MPI implementations on many platforms, due to the complex matching rules in MPI. Even in hardware implementations of MPI matching, the linear traversal of the posted receive queue combined with an interlock between posted and unexpected receive queues, means that there is a dependency between incoming MPI messages. One-sided messages are generally independent, and the dependencies (ordering requirements between two messages in a memory barrier style) are explicit in the program and handled without complex dependencies.

In addition to the potential performance advantage, one-sided also supports applications which have proved to be difficult to implement with message passing. The graph-based informatics applications emerging in the HPC environment pose a problem for message passing implementations, as their communication pattern is determined by the underlying data structure, which is not easily partitioned. Communication with a large number of random processes is common, and the receiving process frequently can not determine which peers will send data. Further, unlike physics codes with iterations of well defined computation and communication phases, many informatics applications do not have well defined computation/communication phases.

For many classes of applications, the one-sided communication paradigm offers both improved performance and easier implementation compared to message passing, even on current hardware with limited performance difference between one-sided implementations

[2]The community is split as to whether Active Messages, in which the sending process causes code to be executed on the target side, is a one-sided interface. Because Active Messages require the main processor to be involved in receiving the message, we do not consider it a one-sided interface.

and MPI. However, there are also application classes in which there is not an advantage to using one-sided communication over message passing, and in which one-sided communication may require more complexity than message passing. Complicating matters further, the common implementations of the one-sided paradigm each have drastically different performance characteristics, and an algorithm which maps well to one implementation may not map well to another implementation.

Therefore, there are a number of issues which must be understood within the one-sided paradigm:

- What features must an implementation of the one-sided communication paradigm provide in order to be useful?
- Which differences between existing one-sided implementations causes one implementation to be suitable for a given application, but another one-sided implementation to be unsuitable for the same application?
- Which applications lend themselves to the one-sided communication paradigm?
- Are there applications in which it is not logical to use the one-sided communication paradigm?

This thesis attempts to answer these questions and provide clarity to a piece of the puzzle in the search for a better programming model for future systems and applications. If, as the author believes, there will not be one dominate programming model on future architectures, but a number of models from which application writers must choose, this thesis is intended to provide guidelines for the applicability of the one-sided communication paradigm for new applications.

4. Contributions

This thesis makes a number of contributions to the high performance computing research area, particularly within the space of communication paradigms. In particular:

- A taxonomy of the one-sided communication space, including the the characteristics which differentiate current one-sided implementations.

- A taxonomy of the requirements on applications which utilize one-sided communication.
- Three case studies which verify both the taxonomy of the one-sided communication space and applications which utilize the one-sided interface.
- A unique, high performance implementation of the MPI one-sided communication interface, implemented within Open MPI.

The remainder of the thesis is organized as follows. Chapter 2 presents background information on a number of subjects frequently referenced in this thesis. In particular, current HPC architectures, popular communication paradigms, the one-side communication interface, Open MPI, and the Parallel Boost Graph Library are discussed.

Chapter 3 first presents a taxonomy of the one-sided communication space, and discusses which features differentiate current implementations. It then proposes a taxonomy of applications which are well suited to the one-sided communication model, which is useful for future application developers in choosing the appropriate communication model. Chapters 4, 5, and 6 present detailed case studies of three applications with very different communication characteristics, in terms of the previously discussed taxonomies. The case studies validate the previous discussion and reveal a number of critical insights into the communication space.

Chapter 7 discusses Open MPI's implementation of the MPI one-sided communication interface, which was developed by the author during early research into this thesis. The implementation is unique in its handling of high message loads in a single synchronization period and in taking advantage of the unique synchronization mechanism of MPI's one-sided interface.

Chapter 8 presents the conclusion of this thesis. This includes an analysis of the features required for a complete one-sided communication framework which is suitable for a wide class of applications, as well as an analysis of other potential message passing replacements based upon lessons learned from the case studies.

CHAPTER 2

Background and Related Work

A number of communication paradigms have been proposed since the emergence of distributed memory HPC systems, including message passing, one-sided, and asynchronous message handling. Each paradigm has a number of trade-offs in performance and usage, which can vary greatly based on the underlying network topology. This chapter provides an introduction to each communication paradigm, as well as details on a number of implementations of the one-sided communication paradigm. In particular, the MPI one-sided communication interface, Cray SHMEM, and ARMCI are presented. Two software packages used extensively during the development of this thesis, Open MPI and the Parallel Boost Graph Library, are then described in detail. The chapter begins, however, with an overview of the current and future state of HPC system architectures.

1. HPC System Architectures

While the commodity HPC market has a wide variety of offerings for processor, memory, and network configurations, the basic system architecture has a number of similar traits:

- A small number (2–4) of processors, each with a small number of cores, although the number of cores is growing.
- A high speed communication fabric supporting OS bypass communication.
- A large amount of memory per core (1–4 GB), although the amount of memory per core is decreasing.

Until recently, a majority of the performance increase in processors has been obtained by increases in the chip's clock rate. Fabrication improvements also allowed for improvements in processor performance through techniques such as pipelining, out-of-order execution, and

superscalar designs. Clock frequencies have largely stabilized due to power and heat constraints that are unlikely to be solved in the near future. Numerous studies have shown that without architectural and programming changes, there is little further to be gained through ILP. The number of transistors available on a die, however, continues to grow at roughly Moore's Law: doubling every 18 months. These constraints have lead processor architects toward multi-core and chip multi-threading processor designs. Both designs increase the computational abilities at the processor at a much higher rate than the memory system improves, leading to an imbalance likely to hurt application performance.

Currently, both Intel and AMD offer quad-core processor designs [**1, 47**]. In high performance computing installations, dual socket installations are the most common form factor, leading to eight computational cores on two sockets. Memory bandwidth has not been scaling at the same pace as the growth in cores, leading to a processor with large computational power, but with less ability to access memory not in cache.

High speed communication systems utilized on modern systems share a number of traits. They generally reside on the PCI Express bus, away from the processor and memory. In order to bypass the kernel when transferring data, the networks must maintain a copy of the user process's virtual to physical memory mapping. It must also ensure that pages are not swapped out of memory when the pages will be used in data transfer. This causes a problem for many HPC networks; they must either receive notification from the kernel whenever the page tables for a process are changed or they must use memory registration to prevent any page used in communication from being moved [**32**]. On Linux, the first option requires a number of hard to maintain modifications to the core of the memory subsystem in the kernel. The second option is more generally chosen for commodity interconnects. Some, like InfiniBand [**45**], require the user to explicitly pin memory before use. Others, like Myrinet/MX [**62**], hide the registration of memory behind send/receive semantics and use a progress thread to handle memory registration and message handshaking. Networks are beginning to move to the processor bus (QPI or HyperTransport) and the PCI Express standard is beginning to support many of the coherency features currently lacking, so it is unclear how these issues will evolve in coming processor and network generations.

2. Communication Paradigms

Three of the most common explicit communication paradigms are message passing, one-sided, and asynchronous message handling. Distributed memory systems have been designed to exploit each of the paradigms, although message passing currently dominates the HPC environment. There are multiple implementations of each paradigm, and this section discusses the paradigm rather than details of any one implementation. Ignored in this section are collective communication routines, which are generally available as part of any high quality HPC communication environment. To help motivate the discussion, a nearest neighbor ghost cell exchange for a one-dimensional decomposition is presented in each paradigm.

2.1. Message Passing. In the message passing communication paradigm, both the sending and receiving processes are explicitly involved in data transfer. The sender describes the data to be sent as well as the destination of the message. The receiver describes the delivery location for incoming messages and can often choose to receive messages out of order based on a matching criteria. Communication calls may be blocking or non-blocking, often at the option of the application programmer. When calls are non-blocking, either the subset of the message which could be transferred is returned to the user or a completion function must be called later in the application to complete the message. Message passing interfaces may buffer messages or may require the application provide all buffer space.

The Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) [**86**] are the most popular examples of message passing in HPC. Traditional networking protocols such as TCP [**22**] and UDP [**71**] could be considered examples of message passing, although they lack many of the features found in MPI and PVM. In addition, most high speed networking programming interfaces, such as Elan [**72**], Myrinet Express [**62**], Open Fabrics Enterprise Distribution [**45**], and Portals [**16**] all provide some level of message passing support.

Figure 1 demonstrates a ghost cell exchange using the message passing paradigm. While the API presented is fictitious, it demonstrates features available in advanced message passing implementations. Remote endpoints are often specified using identifiers based on the

```
double data[data_len + 2], *local_data;
local_data = data + 1;

/* fill in array with initial values */
while (!done) {
    send(local_data, 1, sizeof(double), comm_world, left_tag, my_rank - 1)
    send(local_data + data_len - 1, 1, sizeof(double), comm_world,
         right_tag, my_rank - 1)
    recv(data, 1, sizeof(double), comm_world, left_tag, my_rank - 1);
    recv(data + data_len + 1, 1, sizeof(double), comm_world, left_tag, my_rank + 1);

    /* compute on array */
}
```

FIGURE 1. Nearest neighbor ghost cell exchange using message passing.

parallel job, rather than physical addressing, making it easier to write applications which can run on a variety of machines. Communication may be separated based on contexts, or unique communication channels, which allow different subsets of the application to communicate without conflicting with each other. Finally, tags are used to ensure messages are delivered to the correct location, regardless of arrival order.

2.2. One-Sided Communication. In the one-sided communication model, only one process is directly involved in communication. The process performing communication (the *origin* process) can either send (*put*) or receive (*get*) data from another process (the *target*). Both the origin and target buffers are completely described by the origin process. From the application writer's point of view, the target process was never involved in the communication. A one-sided interface may put restrictions on the remote buffer, either that it be specially allocated, registered with the communication library, or exist in a specific part of the memory space. While put/get form the basis of a one-sided interface, most interfaces also provide atomic synchronization primitives.

Example one-sided interfaces include MPI one-sided communication, Cray SHMEM, and ARMCI, all of which will be discussed in more detail in Section 3. To implement without the use of threads or polling progress calls, all three require significant hardware and operating system support. Figure 2 demonstrates the ghost cell exchange using one-sided

```
double data[data_len + 2], *local_data;
local_data = data + 1;

/* fill in array with initial values */
while (!done) {
    put(local_data, data, sizeof(double), my_rank - 1);
    put(local_data + data_len - 1, data + data_len + 1, sizeof(double), my_rank + 1);
    barrier();

    /* compute on array */
}
```

FIGURE 2. Nearest neighbor ghost cell exchange using one-sided.

communication primitives. In this example, it is assumed that global data members, such as `data`, are allocated at the same address on each process. Most implementations have either a mechanism for making such a guarantee or provide an addressing scheme suitable for global communication. The `barrier()` call also varies greatly between implementations, but is generally available to guarantee the network has completed all started transfers before the application is able to continue. Unlike the message passing example where synchronization is implicit in the receiving of messages, synchronization is explicit in one-sided operations.

2.3. Asynchronous Message Handling. Asynchronous message handling is useful where the data being transferred is irregular and the sender does not know where to deliver the message. For example, an algorithm walking a dynamic graph structure will send messages to random neighbors based on graph structure that can not be determined before execution time. Rather than explicitly receiving each message, as in message passing, a pre-registered handler is called each time a message arrives. The handler is responsible for directing the delivery of the message and potentially sending short response messages.

Active Messages [**56**] is the best known example of the event or callback based communication paradigm, and is frequently cited as an option for future programming interfaces. The concept has also been extended into kernel-level delivery handlers with ASHs [**91**]. Finally, the GASNet project [**13**], which is used by the Berkeley UPC [**55**] and Titanium [**41**]

compilers, provides a combination of active messages with relaxed semantics and one-sided operations.

```
double data[data_len + 2], *local_data;
local_data = data + 1;
volatile int delivered;

void deliver_left(double in) { data[0] = in; delivered++; }
void deliver_right(double in) { data[data_len + 1] = in; delivered++; }

/* fill in array with initial values */
while (!done) {
    send(my_rank - 1, deliver_left, local_data[0]);
    send(my_rank + 1, deliver_right, local_data[data_len - 1]);

    while (delivered != 2) { ; }
    delivered = 0;

    /* compute on array */
}
```

FIGURE 3. Nearest neighbor ghost cell exchange using asynchronous message handling.

Figure 3 demonstrates the ghost cell exchange using an asynchronous message handler. Although two different handlers are used to deliver the left and right peer messages, this could be reduced to a single handler and an extra data field sent in the message to specify the delivery location. The example assumes that no progress function is necessary to receive callbacks from the communication layer. This is true of ASHs, but not necessarily true of other libraries, which provide a poll function from which callbacks will be triggered.

3. One-Sided Communication Interfaces

As this thesis examines the one-sided communication model, further detail on existing one-sided communication interfaces is useful. This section presents three interfaces: MPI one-sided communication, Cray SHMEM, and ARMCI. To help motivate the discussion, a simplified implementation of the inner loop of the HPCC Random Access benchmark (Figure 4) is presented for each interface. Random Access performs random updates on a

large globally distributed array. The kernel is generally considered to perform poorly with message passing but traditionally performed well on high-quality one-sided interfaces. A percentage of the answers may be incorrect, allowing for lock-less implementations of the kernel.

```
long array[len];

for (i = 0 ; i < num_updates ; ++i) {
    idx = get_next_update();
    array[idx] |= idx;
}
```

FIGURE 4. Serial implementation of the Random Access kernel.

3.1. Cray SHMEM. Cray SHMEM [**26, 73**] is arguably the first one-sided library, originally developed for the Cray T3 series of machines. Unlike MPI one-sided, Cray SHMEM is not an official standard and it has seen numerous changes during its lifetime, to better match state of the art hardware. SHMEM provides put and get operations with limited datatype support, as well as compare and swap and fetch and operate atomic operations. Cray's original API consisted of blocking calls, although other vendors later extended the interface to include non-blocking operations. Target memory must either be in the data section or a special symmetric heap. Considerable hardware and software support is required to support Cray SHMEM's loose synchronization rules, and as a result few platforms provide SHMEM support. Currently the Cray platforms, SGI's shared memory platforms and Quadrics-based systems support SHMEM.

Figure 5 presents an implementation of the Random Access kernel using Cray SHMEM. Calls to put and get are blocking, so no additional synchronization calls are necessary between the get and put calls. Unlike MPI one-sided, there is no concept of synchronization epochs, so no synchronization calls are required in the inner loop of the kernel. However, put communication is not required to have completed on the remote side upon the call's return. Two functions, shmem_fence and shmem_quiet, are available to ensure ordering between put calls.

```
long array[len];

for (i = 0 ; i < num_updates ; ++i) {
    idx = get_next_update();
    peer = get_peer(idx);
    offset = get_offset(idx);

    shmem_get(&tmp, 1, peer, array + offset);
    tmp |= idx;
    shmem_put(&tmp, 1, peer, array + offset);
}
shmem_quiet();
```

FIGURE 5. Random Access kernel using Cray SHMEM.

3.2. The Message Passing Interface. The Message Passing Interface (MPI) is a standard for parallel communication developed by the MPI Forum, a collaboration of academic, national laboratory, and industry partners. MPI is actually composed of two standards, MPI-1 [**61, 84**], ratified in 1994, and MPI-2 [**31, 37**], ratified in 1996. MPI-1 provides send/receive and collective communication, as well as run-time environment interrogation. MPI-2 added a number of features, including one-sided communication, dynamic process connectivity, and parallel file I/O. MPI has been implemented for most HPC systems developed in the last 10 years and is the predominant system for parallel applications. A number of MPI implementations exist for commodity cluster systems, including LAM/MPI [**18, 85**], MPICH2 [**4**], and Open MPI [**29**]. These implementations provide excellent performance on a variety of modern HPC platforms.

The MPI-2 standard includes an interface and programming model for one-sided communication. The interface is based around the concept of *windows* of memory that remote processes can access or update. Window creation is a collective operation, and all one-sided communication and synchronization is relative to a given window. Remote addresses are specified as offsets from the base of the specified window, removing the need to determine remote addresses.

MPI one-sided provides three communication calls (put, get, and accumulate) and three synchronization methods (fence, general active target, and passive target). Accumulate

implements a remote atomic operation, although unlike SHMEM it does not return the previous value of the memory location. MPI datatypes are used to describe both the origin and target buffers used in communication, which must meet MPI's loose datatype matching rules. Figure 6 presents an implementation of the Random Access kernel using MPI one-sided communication. The two lock epochs are required so that the MPI_GET operation completes before the `idx` variable is used. MPI_PUT is then used to update the remote value.

```
long array[len];
MPI_Win win;

MPI_Win_create(array, sizeof(long) * len, sizeof(long), MPI_INFO_NULL,
               MPI_COMM_WORLD, &win);
for (i = 0 ; i < num_updates ; ++i) {
    idx = get_next_update();
    peer = get_peer(idx);
    offset = get_offset(idx);

    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, peer, 0, win);
    MPI_Get(&tmp, 1, MPI_LONG, peer, offset, 1, MPI_LONG, win);
    MPI_Win_unlock(peer, win);
    tmp |= idx;
    MPI_Win_lock(MPI_LOCK_EXCLUSIVE, peer, 0, win);
    MPI_Put(&tmp, 1, MPI_LONG, peer, offset, 1, MPI_LONG, win);
    MPI_Win_unlock(peer, win);
}
MPI_Win_free(&win);
```

FIGURE 6. Random Access kernel using MPI one-sided.

3.2.1. *Communication Epochs.* MPI one-sided communication uses the concept of epochs to define when communication can occur and when it completes. All communication calls acting on a window must occur while the window on the origin process is in a *access epoch*. Similarly, the target of those communication calls must be in an *exposure epoch*. The communication calls do not complete until their respective epochs have completed. Epochs are started and completed by the MPI one-sided synchronization calls.

3.2.2. *Communication Calls.* The MPI_PUT and MPI_GET function calls provide basic data movement from one process to another. The MPI_ACCUMULATE call offers the opportunity to perform atomic read-modify-write calls on remote operations using any of the operations that are valid for MPI_REDUCE. For example, MPI_SUM can be used to implement an atomic increment operation. All three calls are non-blocking and are completed by the synchronization routines described in this section. Local completion is guaranteed at the end of the local exposure epoch and remote completion is guaranteed at the end of the remote access epoch. One frequently misunderstood point is that the origin's buffer does not need to be in the buffer described by the window argument, as the window is used only to determine the remote memory location.

3.2.3. *Fence Synchronization.* MPI_WIN_FENCE (Figure 7) involves synchronization between all processes in the given window. No particular synchronization (barrier or otherwise) is implied by a call to MPI_WIN_FENCE, only that all communication calls started in the previous epoch has completed. A call to MPI_WIN_FENCE completes both an exposure and access epoch started by a previous call to MPI_WIN_FENCE. It also starts a new exposure and access epoch if it is followed by communication and another call to MPI_WIN_FENCE. Hints can be used to tell the MPI implementation that the call to MPI_WIN_FENCE completes no communication or that no communication will follow the call. Both hints must be defined globally - if any one process provides the hint, all processes in the window must provide the same hint.

3.2.4. *General Active Target Synchronization.* When global synchronization is not needed because only a small subset of the ranks in a window are involved in communication, general active target synchronization (also known as *Post/Wait/Start/Complete synchronization*) offers more fine-grained control of access and exposure epochs. Figure 8 illustrates the sequence of events for general active target synchronization. A call to MPI_WIN_START starts an access epoch, which is completed by MPI_WIN_COMPLETE. MPI_WIN_START will not return until all processes in the target group have entered their exposure epoch. A process starts an exposure epoch with a call to MPI_WIN_POST and completes the exposure epoch with a call to MPI_WIN_WAIT or can test for completion with MPI_WIN_TEST.

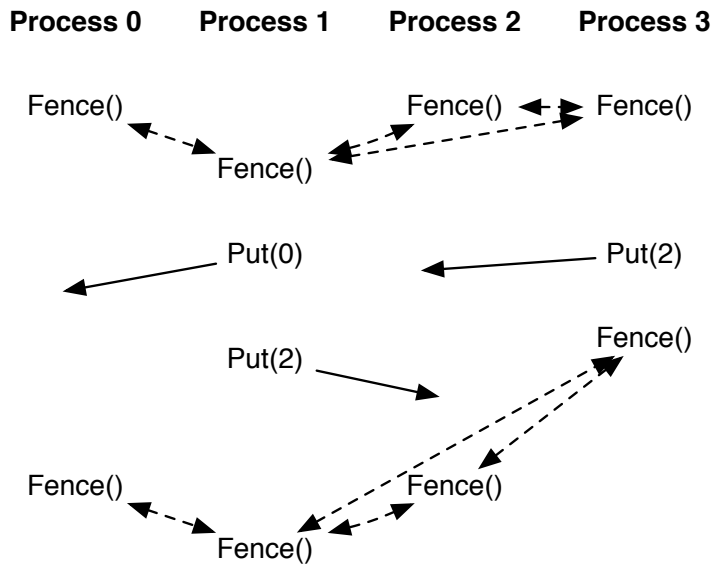


FIGURE 7. Fence Synchronization. All processes are in both an exposure and access epoch between calls to MPI_WIN_FENCE and can both be the origin and target of communication. Solid arrows represent communication and dashed arrows represent synchronization.

Exposure epoch completion is determined by all processes in the origin group passed to MPI_WIN_POST having completed their access epochs and all pending communication having been completed.

3.2.5. *Passive Synchronization.* To implement true one-sided communication, MPI provides MPI_WIN_LOCK and MPI_WIN_UNLOCK. The origin side calls MPI_WIN_LOCK to start a local access epoch and request the remote process to start an exposure epoch (Figure 9). Communication calls can be made as soon as MPI_WIN_LOCK returns, and are completed by a call to MPI_WIN_UNLOCK. Lock/Unlock synchronization provides either shared or exclusive access to the remote memory region, based on a hint to MPI_WIN_LOCK. If shared, it is up to the user to avoid conflicting updates.

3.3. ARMCI. ARMCI [**65**] is a one-sided interface originally developed as a target for more advanced protocols and compiler run-times, notably the Global Arrays project [**66, 67**]. ARMCI is notable for its advanced support for non-contiguous memory regions, which removes the need for upper layer libraries to pack messages before sending, typically required

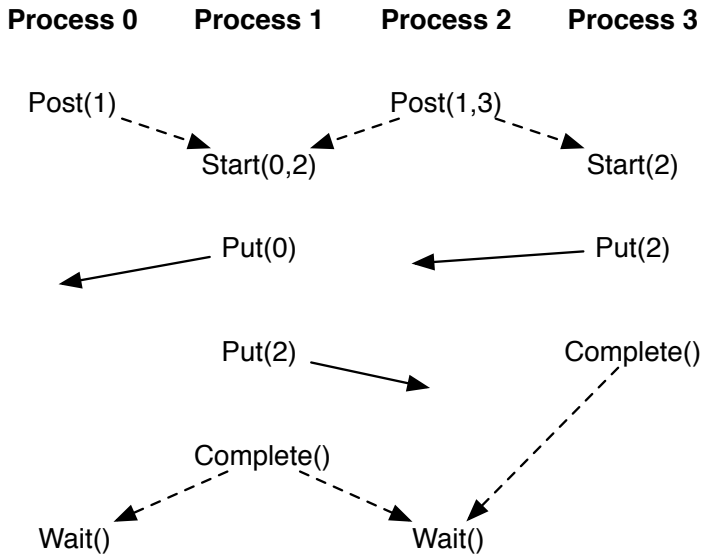


FIGURE 8. General Active Target Synchronization. A subset of processes in the window may individually start a combination of access and exposure epochs. Solid arrows represent communication and dashed arrows represent synchronization.

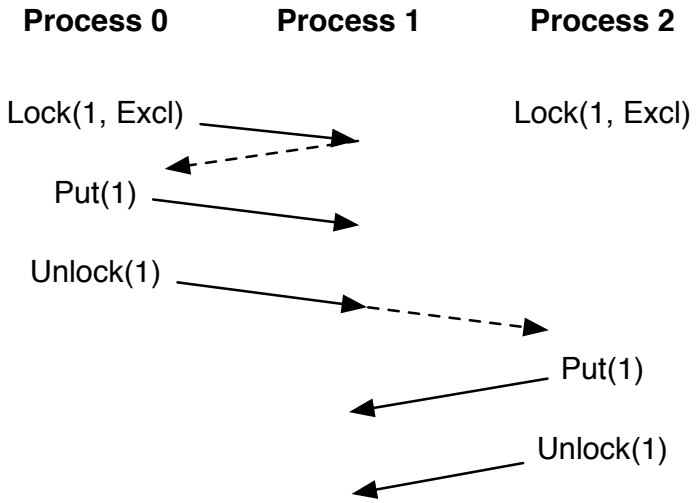


FIGURE 9. Passive synchronization. Communication between the origin and target can not begin until an acknowledgement is received from the passive process, although no user interaction is necessary to generate the acknowledgement. Solid arrows represent communication and dashed arrows represent synchronization.

to achieve performance in other libraries. Communication calls include put, get, accumulate, and read-modify-write operations. Upon return from a write call, the local buffer is no longer in use by ARMCI and may be modified by the application. Data is immediately available upon return from a read call. A fence operation provides remote completion of write operations.

```
long *array[nprocs];

ARMCI_Malloc(array, sizeof(long) * len);

for (i = 0 ; i < num_updates ; ++i) {
    idx = get_next_update();
    peer = get_peer(idx);
    offset = get_offset(idx);

    tmp = ARMCI_GetValueLong(array[peer] + offset, peer);
    tmp |= idx;
    ARMCI_PutValueLong(tmp, array[peer] + offset, peer);
}
ARMCI_Barrier();
```

FIGURE 10. Random Access kernel using ARMCI.

Figure 10 demonstrates the Random Access kernel in ARMCI. Since there is no opportunity for non-contiguous transfers in the kernel, the strengths of ARMCI are not shown in this example. However its true one-sided semantics and low synchronization requirements lead to a straight-forward implementation of the kernel.

4. Related Software Packages

Two software packages, Open MPI and the Parallel Boost Graph Library, were modified extensively as part of this thesis. A brief overview of both software packages is presented.

4.1. Open MPI. Open MPI is a recent MPI implementation, tracing its history to the FT-MPI [**28**], LAM/MPI [**85**], LA-MPI [**34**], and PACX-MPI [**50**] projects. The project is developed as a collaboration between a number of academic, government, and commercial institutions, including Indiana University, University of Tennessee, Knoxville, University of Huston, Los Alamos National Laboratory, Cisco Systems, and Sun Microsystems.

Open MPI is a complete implementation of the MPI-1 and MPI-2 standards designed to be scalable, fault tolerant, and provide high performance in a variety of HPC environments. Open MPI supports a variety of platforms, including clusters running Linux, Mac OS X, and Solaris. It has also been ported to the Cray Red Storm/XT-3/XT-4/XT-5 tightly coupled MPP systems [**8**]. Open MPI's performance on commodity Linux clusters with high speed interconnects is well established [**81, 92**], with optimization work ongoing.

Open MPI utilizes a low-overhead component architecture, the Modular Component Architecture (MCA), to provide abstractions for portability and adapting to differing application demands. There are component frameworks for everything from printing a stack trace to encapsulating the MPI point-to-point and collective semantics. In addition to providing a mechanism for portability, the MCA allows developers to experiment with different implementation ideas, while minimizing development overhead. This flexibility has already resulted in the development of an advanced tuned collectives implementation and the ability to adapt the point-to-point interface to use either network-level or MPI-level match queue searching for send/receive semantics. As shown in previous work [**9**], the MCA provides this flexibility with a minimum overhead, essentially the same as utilizing shared libraries.

MPI communication is layered on a number of component frameworks, as shown in Figure 11. The PML provides MPI send/receive semantics, including message matching and ordering. The BML is a very thin layer that maintains the available routes to a particular peer and handles message scheduling across multiple endpoints to a given peer. The BTL framework provides communication between two endpoints, where an endpoint is usually a communication device connecting two processes, such as an Ethernet address or an InfiniBand port. The design and implementation of the communication layer is described in detail in [**81, 92**].

The BML/BTL design is intended to simultaneously support multiple upper-layer protocols. Presently, this has been shown by supporting both the PML MPI point-to-point communication and a one-sided component. The use of the BML/BTL interface for advanced collective implementations is currently under investigation.

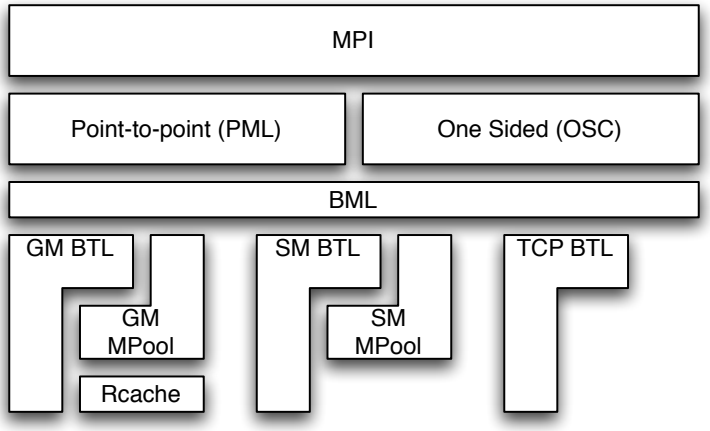


FIGURE 11. Component structure for point-to-point communication in Open MPI.

BTL components provide two communication modes: an active-message style send/receive protocol and a remote memory access (RMA) put/get protocol. All sends are non-blocking, with a callback on local send completion. Sends can either be zero copy or copied into BTL-specific memory before transfer. Receives are all into BTL-provided buffers, with a callback on message arrival. RMA operations provide callbacks on completion on the origin process and no completion callbacks on the target process. All buffers used on both the origin and target must be "prepared" for use by calls to the BTL by higher-level components.

4.2. The Parallel Boost Graph Library. The Parallel Boost Graph Library (Parallel BGL) [**36**] is a high performance generic C++ library for distributed graph computation. The Parallel BGL builds upon the serial Boost Graph Library (BGL) [**82**] and utilizes many of the same algorithms. The parallel abstractions are "lifted" from the serial abstractions where required. [**35**]. Unlike previous graph libraries, the Parallel BGL is not tied to a particular graph data structure but, like the C++ Standard Template Library (STL), is designed to allow algorithms to operate on a variety of graph representations.

4.2.1. *Graph Representations.* The Parallel BGL supports a variety of graph data structures, including adjacency list, adjacency matrix, and a highly space efficient compressed sparse row (CSR) format. The adjacency list type can further be parametrized to use STL

lists, vectors, sets, or multisets for both the vertex list and the adjacency list for a given vertex. Each format offers a different set of performance characteristics, allowing the library user to choose the most efficient structure for a particular application.

The CSR representation, while the most space efficient and generally offering the best performance, must have edges added in a sorted order, meaning that modifying the graph is essentially impossible once it has been created and that generating the graph can be difficult. For example, generating random graphs which previously required constant space may now require linear space.

The adjacency list representation offers less space efficiency than the CSR representation, but is relatively efficient for relatively sparse graphs (dense graphs should use the adjacency matrix representation). Storing the vertices and their adjacencies in vector format is relatively space efficient, while utilizing a list structure allows edge or vertex modification in constant time. A set representation of the adjacency list allows for implicit removal of duplicate edges.

Partitioning of the graph across a distributed memory architecture is automatic if done during the graph constructor, but is explicit if the graph is later modified. A vertex may only be added from the process in which it is to be stored. Similar care must be taken when adding new edges to the graph. During graph construction, a block distribution is used by default, although cyclic, block cyclic, and uniformly random distributions are also available.

4.2.2. *Property Maps.* Property maps associate information with the vertices and edges in a graph, in a format that can be decoupled from the graph format itself. For example, edge weights, rather than being stored in the graph itself, may be stored in an external property map structure. The properties of a vertex or edge may be set via a `put` function or retrieved via a `get` function.

The property map concept allows many serial algorithms to operate on a distributed graph through the *distributed property map*, which implements `put` and `get` operations which operate on both local and remote data. In the common case, when the Parallel BGL is utilizing the MPI process group (process groups are discussed later), the put and get operations

to remote processes do not complete until the next synchronization phase. A resolver function may be added by the algorithm author to cope with collisions in updates which may occur during a synchronization phase. The resolver may contain a significant amount of logic which directs the next computation phase, such as that found in the parallel search version of connected components, or it may be used to implement an atomic operation, such as the summation found in the PageRank algorithm.

As part of our investigation into the one-sided communication paradigm, two property maps were developed: one based on Cray SHMEM and another based on MPI one-sided. Both greatly limit the flexibility of the data storage, currently requiring that data be stored in vectors and that the graph have local vertex identifiers ranging from $0 \ldots N-1$. This limitation will be discussed in detail in Chapters 4 and 5.

4.2.3. *Process Groups.* The Parallel BGL utilizes a distribution library originally developed for the Parallel BGL, but which is sufficiently general to be used elsewhere, the Parallel Processing Library. The Parallel Processing Library is based on the concept of a Process Group, which encompasses the basic functionality found in most parallel applications: a concept of the number of parallel entities (processes, threaded, etc.), the id of "my" entity, and useful collectives in parallel algorithms: all-gather and all-reduce. Further specialization, such as the addition of messaging, is added by refinements of the base concept.

The MPI Process Group refinement exposes send and receive semantics based on the MPI interface and is loosely based on a relaxed Bulk Synchronous Process (BSP) [**83**] model. The immediate process group is currently the most commonly used and default process group in the Parallel BGL. While maintaining the computation/synchronization phases of BSP, it also allows for message transfer and reception before the completion of a computation phase, which can greatly simplify some graph algorithms.

Both a threaded process group, in which the division of parallelism was threads instead of MPI tasks, and a process group based on MPI one-sided, have also been implemented, although neither currently sees use. The MPI one-sided process group was part of the initial work in investigating the one-sided communication paradigm and proved to be too

inflexible for the Parallel BGL abstractions. The Process Group abstraction is too low level and hidden by other structures, such as the property map, for applications to directly take advantage of the one-sided communication paradigm. The MPI one-sided interface also proved insufficient to implement the MPI process group abstraction due to issues with unbounded message sizes during a BSP phase. The MPI one-sided property map combined with the MPI immediate process group has showed much greater promise and is the structure used for the MPI one-sided results presented in later chapters.

CHAPTER 3

A One-Sided Taxonomy

As discussed in Chapters 1 and 2, recent trends in high performance computing are opening the door to paradigms other than message passing. This growth in programming options creates enormous opportunities for programmers to chose the communication paradigm and interface best suited to their application, rather than the one, perhaps two, interfaces previously available on a given platform. Unfortunately, it is not practical for developers to create multiple implementations of a large scale application, each with a different communication paradigm and application level communication abstractions are typically biased towards a particular paradigm. Therefore, it imperative for the high performance computing community to develop a set of guidelines for which application and paradigm combinations are most likely to lead to success.

Traditionally, the answer to the question of when to use a one-sided communication paradigm for application development is whenever message passing is a bad fit. Such an answer is problematic for a number of reasons, not the least of which is that there isn't a good definition of when the message passing paradigm is a good fit. Such "I know it when I see it" methodology may work well for experienced parallel application developers, but generally requires failing at a message passing design before attempting a one-sided design.

For the purposes of this dissertation, we define the one-sided paradigm somewhat narrowly, to include only those interfaces who's primary communication method does not involve any action by the target process. Under such a definition, Active Messages and interfaces based on Active Messages such as GASNet, are not considered to be one-sided interfaces. Unlike the motivating platforms for active-message style interfaces, modern processor are unable to quickly handle interrupts, requiring active messages to be based on a polling or threading model. It is our belief that this polling characteristic substantially

changes application programming style and therefore constitutes its own category of parallel programming paradigms.

We begin by presenting a taxonomy of the one-sided programming paradigm in Section 1. While likely incomplete, we believe that it is sufficient for the driving purpose behind the work: to assist application developers in picking the programming model that best fits their application. Section 2 expands upon the taxonomy from the previous section to discuss the traits which make an application suitable for one-sided communication paradigms. Finally, we present a discussion of when we believe using one-sided communication is appropriate, combining the information presented in Sections 1 and 2.

1. One-Sided Paradigm

We assume that all one-sided interfaces provide the most basic one-sided functionality, put and get operations capable of storing and loading, respectively, data to a remote process. Similar to the basic store and load instructions of a processor used to interact with local memory, these two operations form the backbone of the interface for interacting with remote memory. Differentiation characteristics include blocking behavior, atomic operations, synchronization, memory utilization, and collective operations.

1.1. Blocking vs. Non-Blocking. Put and get operations may be either blocking or non-blocking. Blocking calls present an easier implementation path for the one-sided library and require fewer resources from the network interface card and network, as there can be many fewer operations in flight at any given time. However, a non-blocking interface presents the possibility of overlapping multiple communication operations, particularly with get operations, which require a round-trip to remote memory to complete. As shown in Chapter 5, the difference between blocking and non-blocking operations in terms of application performance can be substantial. However, in addition to the cost of added implementation complexity, non-blocking interfaces may greatly increase the complexity of applications, due to the required state management (request tracking, barrier synchronization, etc.).

1.2. Atomic Operations. Local load and store operations, while sufficient for many serial applications, are insufficient for applications requiring synchronization. Similarly, while put and get are sufficient for a small number of applications (see the HPCCG case study in Chapter 6), many applications require a richer set of operations in order to implement remote synchronization. Four different synchronization mechanisms appear in one-sided paradigms:

Atomic Operation: A mathematical operation (add, multiply, max, min, etc.) atomically updates the target memory location.

Atomic Fetch and Operate: A mathematical operation (add, multiply, max, min, etc.) atomically updates the target memory location, with the original value returned to the origin process.

Compare and Swap: A new value is atomically swapped with the current value of an address in the target memory if and only if the current value at the target is equal to a specified value.

Lock/unlock: Similar to a threading mutex, provides blocking synchronization for a memory region.

Processors often provide one or more of the above synchronization mechanisms for interacting with local memory. Because the memory system is incapable of performing mathematical operations, the processor must be involved in the operation and the Atomic Operation and Atomic Fetch and Operate mechanisms are equivalent. It has been proved that Atomic Fetch and Operate and Compare and Swap (in addition to Load Locked/Store Conditional, a mechanism not found in one-sided interfaces) are syntactically equivalent and evidence suggests that they are also performance equivalent for local updates. [**39, 90**]

The latency between processor and memory is much smaller than the latency to a remote memory location, and NICs are frequently capable of performing integer mathematical operations. Because the NIC is capable of performing the operation on the target, a Compare and Swap, an Atomic Operation, and an Atomic Fetch and Operate are frequently implemented differently, with very different performance characteristics. As will be shown

in the PageRank case study of Chapter 5, there can be a drastic performance difference based on which atomic primitives are available in a given one-sided interface.

1.3. Synchronization. One-sided interfaces frequently differ in how they handle communication synchronization. For example, Cray SHMEM applications can be written with no explicit synchronization calls, but MPI one-sided requires complex epoch synchronization (See Chapter 2) for all communication calls. To improve performance, one-sided interfaces which do not require explicit synchronization in the general case frequently relax completion semantics of operations which do not require a reply (put, Atomic Operation, etc.), such that completion of the call only guarantees local completion, but not remote completion. Such interfaces then require a synchronization call to guarantee remote completion.

While there appears to be a general consensus that unsynchronized interfaces such as Cray SHMEM are more desirable, they lack optimizations such as scheduling and coalescing, which are available to interfaces in which the synchronization points in the code are explicit. On platforms with a low relative performance of one-sided to message passing (see Section 1.5), these optimizations may be critical to application performance. Non-blocking operations provide similar opportunities, since the completion point for the non-blocking operations acts as a synchronization point, in that the communication library can rightfully assume a completion call will be called at some point in the future, before data delivery is required for application correctness.

1.4. Target-Side Memory. One-sided implementations limit message processing on the target side of an operation to a specific region of memory. Cray SHMEM, for example, limits the target addresses to an area known as the symmetric heap, while MPI limits target addressing to a given Window. In both cases, it is erroneous to send messages which target memory addresses outside of the specified range.

Allocation of the target side memory region also differs among one-sided interfaces. Cray SHMEM and ARMCI require a special allocator be used for allocating memory which will be the target of one-sided operations. The ARMCI allocator can be used in either a collective or single-process mode, while the Cray SHMEM allocator is pseudo-collective: the

call itself is not collective, but every process must make the same number of calls with the same arguments in the same order. This tends to lead to code like that shown in Figure 1. MPI, on the other hand, uses the concept of a Window to allow communication into a memory range which has already been allocated.

```
Allreduce(&nrow, &mnrow, 1);
double *r = (double*) shmalloc(sizeof(double) * mnrow);
Allreduce(&ncol, &mncol, 1);
double *p = (double*) shmalloc(sizeof(double) * mncol);
```

FIGURE 1. Memory allocation pattern using Cray SHMEM

Target addresses may be specified as either a virtual address (Cray SHMEM or ARMCI) or an offset from a starting address (MPI). MPI provides the ability to create many different memory regions (Windows). A *displacement* from the start of the window, multiplied by a value specified at Window creation time, is used to generate a virtual address on the target. The use of the symmetric heap and virtual addresses makes Cray SHMEM generally straight-forward for homogeneous architectures. The MPI design, with displacements based on a window-creation time constant, offers greater support for heterogeneous applications. For example, on platforms with different padding rules, the storage of a structure may require a differing amount of space. By specifying the padded structure size as the displacement unit during window creation time, this processor-specific artifact is avoided.

1.5. Relative Performance. While generally not a property of a given one-sided interface or the one-sided paradigm as a whole, performance relative to message passing is an important property in the success of a given implementation. Latency and, even more so, message rate, are critical to the usability of a one-sided interface. One-sided interfaces encourage users to make more communication calls with smaller buffers when compared to the message passing paradigm. When a given platform is message rate limited and there is not a considerable difference between the message passing and one-sided message rate, one-sided implementations suffer in head-to-head comparisons, as they are throttled by their higher message counts.

Going forward, one-sided implementations will generally have lower latency and higher message rates than MPI is capable of delivering. Software MPI implementations are becoming constrained by the need to walk a linked list of posted receives on every incoming message, as well as walk another linked list of unexpected messages whenever a new receive is posted. Walking a linked list is inherently unfriendly to cache structures, and therefore takes much longer than a one-sided implementation in which the delivery address is known without any list walking. Hardware implementations of MPI fair better in comparison, but still require more processing time per incoming message, because the posted receive list (now a hardware construct) still must be locked and walked atomically for incoming messages.

1.6. Collectives. Collective operations, such a broadcast and reduce, are generally fundamental to at least part of any distributed algorithm. For example, iterative solvers generally perform a reduction at the end of every step to check if the residual has dropped below a set threshold and the algorithm has completed. Broadcasts are frequently used to load initial state during application start-up. While basic collective routines can be easily implemented using message passing, active messages, or one-sided, high performance collective routines require a good deal of optimization, including machine-specific tuning. [**3, 51, 70**]

Therefore, while collective routines are not strictly necessary as part of a one-sided interface, it is useful for application programmers to rely on the interface to provide collective operations with clear semantics and which are efficiently implemented for the particular machine in use. In all case studies we present, we assume that collective routines tuned for the target machine are available, even when using Cray SHMEM, which does not natively provide such routines.

1.7. Summary. Table 1 compares the three one-sided interfaces discussed in Chapter 2 based on the taxonomy presented in this section. The interfaces differ in ways not described in this section, for example in how strided arrays of data would be communicated. However, we believe for common use cases these details are not critical to the success of an application

in using any particular one-sided communication interface. Further, while performance is critical to the success of a particular implementation, it is difficult to compare interface performance and we therefore make no attempt to do so.

	Blocking	Atomic Operations	Synchronization	Target Side Memory	Collectives
Cray SHMEM	Blocking, non-blocking put	Atomic Fetch and Operate, Compare and Swap, Lock-/Unlock	Implicit, optional write barrier	Symmetric Heap	Barrier and Broadcast
MPI One-Sided	Non-blocking	Atomic Operate	Explicit communication epochs	Windows, target-computed offsets	Complete MPI collectives
ARMCI	Both	Fetch and Operate, Atomic Fetch and Operate, Lock/Unlock	Explicit	Special allocators	Generally provides MPI

TABLE 1. Analysis of Cray SHMEM, MPI one-sided, and ARMCI according to proposed taxonomy.

The differences shown in Table 1 can be traced to the original goal of the interfaces. Cray SHMEM was designed to expose the unique performance characteristics of the Cray T3D and later the SGI shared memory machines, and focused on exposing the minimalistic interface for remote memory access. The relative latencies between local and remote memory access was so small that a blocking interface was sufficient. The MPI one-sided interface had to be supported on both high-end machines and Ethernet clusters. The interface, particularly the explicit synchronization epochs, result from this lowest common denominator design methodology. ARMCI, on the other hand, evolved to support the needs of the Global Arrays project and the limited set of applications which use Global Arrays. While the explicit synchronization fulfills many of the same goals of the MPI one-sided, it is seen as less invasive, as it conforms to the general usage patterns of Global Arrays.

As one-sided interfaces become better supported and more widely used, it is likely that the interfaces will continue to evolve. Cray SHMEM has evolved as each hardware platform is released, to best exploit the capabilities of new hardware. MPI one-sided has

remained relatively static since its introduction in MPI-2, although the MPI Forum is currently discussing changes for the MPI-3 standards effort. These changes include adding a richer set of atomic operations, including Atomic Fetch and Operate and Compare and Swap operations, and are discussed in Chapter 8.

2. One-Sided Applications

Like any other tool, even the best one-sided implementation will not work well if used incorrectly. It is our belief that there is no silver bullet of communication paradigms, and therefore not all applications fit a one-sided communication paradigm. This section presents a number of factors critical in deciding whether the one-sided communication model is appropriate for a given application. In addition to raw performance, ease of implementation must be taken into account.

2.1. Addressing Scheme. One-sided communication paradigms use origin generated addressing for target side buffers. The origin generated address may be an actual virtual address, as with Cray SHMEM, or a region id and offset, as with MPI one-sided and window-based addressing. In either situation, the origin must be able to generate the correct address with a minimal amount of space and addition communication. Sparse data structures, such as linked lists, may be impractical as targets unless the data stored in each element is much larger than the combination of a process identifier and pointer (which generally total 12 or 16 bytes).

For example, while the algorithms described in the first two case studies (Chapters 4 and 5) fit the remaining requirements quite well, the graph structure in use may prohibit reasonable use of a one-sided paradigm. Array-based structures provide an ideal storage mechanism for both applications when used with one-sided, as the address is a base address combined with a well-known offset, and the node id and offset can typically be encoded in 8 bytes or less. List-based structures, on the other hand, allow for a much more dynamic graph, but at a high data storage cost. In the case of page rank, 12 bytes would be required to store a remote address which contains 8 bytes of interesting data.

2.2. Completion Semantics. Unlike many other communication paradigms, one-sided paradigms are unique in that they generally do not provide notification to the target process that data has arrived (other than the change in the contents of the target memory location). For many applications, there is a well defined communication phase and a well defined computation phase. In these cases, the required synchronization calls (Section 1.3) will generally provide sufficient completion notification. The PageRank and HPCCG case studies which follow both fall into this category.

On the other hand, many applications are designed to react to incoming messages (discrete event simulators, database servers, etc.). The one completion semantic available from one-sided interfaces—data delivery—is often insufficient as it leads to polling memory for messages. In addition to causing numerous reads from memory due to the inability of NICs to inject new data into a processor's cache, polling requires the process to be actively using the processor. While the performance implications of hard polling for message arrival are not severe, the power usage is undesirable. Further, due to data arrival ordering issues within a message and strange interactions with modern cache and memory structures, such schemes can be fragile and error prone.

2.3. Work / Message Independence. A number of algorithms depend upon structures such as work queues, stacks, and linked lists. Similar to the problems faced implementing such structures in a multi-threaded environment using lock-less operations (see Section 1.2), implementing the structures in a one-sided implementation proves difficult. The structures are generally trivial to implement using an active message paradigm and relatively straight forward when using message passing. One-sided, however, presents both design and scalability problems. Because message delivery is based on origin-side addressing, the delivery address must be known prior to delivery at the target node. For work queues, the problem can be solved by per-peer circular buffers, although there are obvious scaling problems associated with per-peer resources. Stacks could potentially be implemented using a compare and swap, although the performance is likely to suffer if there is any contention, due to the high cost of multiple round trips for a single message. Linked

lists are the most problematic, as there are few viable solutions outside of remote locks protecting the list.

In some cases, such as the connected components algorithm presented in Chapter 4, the algorithm can be modified to eliminate the use of a work queue, potentially at the cost of slightly more work. In some cases, such as a command driven data server, a work queue may be unavoidable. In such cases, it is unlikely that one-sided will be a viable model for implementation. Such an issue is one of the reasons that existing applications written using a message passing or active message paradigm have historically been unsuccessfully converted to using a one-sided paradigm.

2.4. Summary. Three case studies are presented in later chapters verifying the one-sided taxonomy presented in Section 1 and the application interface requirements presented in this section. Table 2 presents the three case studies according to the topics discussed in this section.

	Addressing Scheme	Completion Semantics	Work Independence
Connected Components	Offset from global array start	Single barrier	Work follows graph structure
PageRank	Data-structure dependent, generally offset from global array start	Barrier per iteration	Work follows graph structure
HPCCG	Per-peer offset into array	Completion with small set of peers	Work based on data partitioning

TABLE 2. Analysis of Connected Components, PageRank, and HPCCG applications according to proposed taxonomy.

The Connected Components and PageRank problems both involve global communication: at each iteration of the algorithm, it is likely a process will communicate with a high percentage of other processes. This communication pattern mitigates the cost of global synchronization calls, as the impact of such a call is low if synchronization is needed with a large percentage of processes. If, on the other hand, synchronization is only needed with a small number of processes, as is the case with HPCCG, the cost of global synchronization calls is much higher. For interfaces which provide only implicit synchronization, this trait may pose undue performance problems.

3. Conclusions

This chapter presents a taxonomy for one-sided interfaces, as well as a set of guidelines for determining whether an application is suitable to use with a one-sided paradigm. Unfortunately, it is unlikely that any one communication paradigm will be sufficient for all applications, hence the need for a better understanding of the issues associated with any given model.

Our analysis of the applications and communication interfaces attempts to categorize implementation issues with a given one-sided interface according to the following breakdown:

Paradigm: Issues occurring at the paradigm level are not related to a particular interface or library. An example of such an issue is the completion semantics issue discussed in Section 2.2.

Interface: Interface issues include those related to a particular interface (Cray SHMEM, MPI one-sided, etc.), and include details like the availability of blocking versus non-blocking calls or available synchronization primitives.

Implementation: Implementation issues include usability or performance shortcomings due to a given implementation of an interface. The poor performance of some MPI one-sided implementations, particularly as message load increased, is one significant implementation issue.

Hardware: Issues related to a particular hardware platform. For example, the Red Storm platform used for SHMEM results in the case studies have a limited number of outstanding operations (2048) and a comparable message rate for one-sided and message passing.

MPI one-sided implementations, in particular, are notoriously immature, which is likely due to the small user community and poor implementation options available with current hardware offerings. As part of early work on this dissertation, we have implemented the MPI one-sided interface within Open MPI, which is described in Chapter 7. Our intention in categorizing implementation issues using this breakdown is to define the severity of a particular problem. For example, hardware issues are likely to cause problems on current

generation hardware, but may very well disappear in the next 12–24 months. Paradigm issues, however, are so severe as to suggest the one-sided paradigm is permanently unable to support a given application.

To validate both our one-sided taxonomy and the application evaluation criteria, we present three case studies in the following chapters: A Connected Components algorithm implementation (Chapter 4), The ubiquitous PageRank algorithm (Chapter 5), and an implicit finite element solver (Chapter 6). The MPI message passing interface, the Cray SHMEM one-sided interface, and the MPI one-sided interface are compared in each case study, including implementation and performance results where possible.

CHAPTER 4

Case Study: Connected Components

Identifying the connected components, the maximally connected subgraphs, of a graph is a challenging problem on distributed memory architectures. It is also an important concept for informatics, both directly to identify connections within data and indirectly to support other algorithms by breaking data into smaller independent pieces.

Connected components presents scaling and performance challenges to distributed memory architectures due to the excessive communication needed in most algorithms and, more importantly, the interdependence of data, making communication overlap difficult. The random communication patterns and short messages of connected component algorithms would appear to make it an ideal candidate for one-sided communication models, and results presented in Section 3.1 support such an assertion.

This chapter begins by presenting an overview of three common parallel algorithms in Section 1. An analysis of the communication properties and their applicability to the one-sided communication paradigm is presented in Section 2. Finally, an analysis of the implementation of a Bully Connected Components algorithm using Cray SHMEM and MPI one-sided is presented in the context of Section 2 in Section 3.

1. Connected Component Algorithms

Identifying connected components can be efficiently implemented as a series of depth-first searches, with the visitor identifying the component membership of each newly discovered vertex. While depth-first search is efficient in serial applications, parallel performance is more difficult [**35**]. A number of distributed memory connected component algorithms have been proposed, including hook and contract algorithms by Awerbcuh and Shiloach [**6**] and Shiloach and Vishkin [**79**], and random contraction algorithms by Reif [**75**]

and Phillips [**69**]. Kahan's algorithm utilizes a parallel search[1] combined with Shiloach-Vishkin on multi-threaded shared memory platforms [**11**]. The Bully algorithm refines Kahan's algorithm to remove hot-spots on shared memory platforms.

A simple, high performance distributed memory algorithm based on the Bully algorithm but influenced by Shiloach-Vishkin and Kahan's algorithm to reduce communication costs is used to motivate discussion of one-sided communication paradigms. The Parallel BGL Parallel Search algorithm, on which the work presented in this chapter is based, is heavily influenced by both the Kahan and Bully algorithms. All three algorithms are presented in further detail, to motivate the discussion of one-sided communication in Section 2.

1.1. Kahan. Kahan's algorithm is designed for massively multi-threaded shared-memory architectures like the Terra MTA [**2, 20**] and Cray XMT [**25**]. The algorithm optimizes for high levels of parallelism, 5,000 threads for the MTA and upwards of 256,000 threads on the XMT), over limiting random communication patterns.

Kahan's algorithm labels the connected components of a graph in three phases:

(1) Parallel searches are started from every vertex in the graph, marking unvisited vertices as being in the previous vertex's component. If a vertex has already been visited and is marked as belonging to another component, the two components are entered into a hash table of "component collisions".
(2) The Shiloach-Vishkin algorithm is used to find the connected components of the graph consisting of all the collision pairs in the hash table generated by the first step.
(3) Parallel searches are started from the component leader (the vertex that originally started the given component in step 1) for the "winning" component of all the collision pairs, marking the vertices in the graph as belonging to the winning component.

[1]A parallel search is similar to a breadth-first search, but without the requirement that each "level" of the graph be visited before moving on to the next level. In general, it scales well on shared memory, distributed memory, and multi-threaded platforms.

Kahan's algorithm is impractical without refinement on distributed memory machines, due to the need for a global hash table. Such data structures prove impractical at a large scale for distributed memory machines. The hash table also proved problematic for larger scale MTA and XMT machines, as collisions when inserting into the hash table caused a high degree of memory hot-spots, which caused severe performance degradation on the cache-less platforms. These performance characteristics lead to the Bully algorithm, a refinement of Kahan's algorithm.

1.2. Bully. The Bully algorithm starts with the same principle as Kahan's: a large number of parallel searches marking component membership. The algorithms diverge in their handling of a vertex that appears to belong in two different components. Unlike Kahan's algorithm, in which resolution of the conflict is deferred and an external global data structure is used to store the collisions, the Bully algorithm immediately resolves the collision and allows only the winning parallel search to continue.

When a search discovers a vertex that already belongs to different component, the components are compared and a "winning" component is selected.[2] If the parallel search was started by the losing component, it stops all further searching. If the parallel search was started by the winning component, it becomes the "bully" and overwrites the previous component information with its own and continues searching.

While the Bully algorithm does not utilize a global data structure subject to hot-spotting like Kahan's, it does require a rich set of inexpensive synchronization or atomic operations. The Terra MTA and Cray XMT on which the algorithm was developed use Full-Empty bits for read and write synchronization at virtually no extra cost over traditional reads and writes.

1.3. Parallel BGL Parallel Search. The Parallel BGL provides two connected components algorithms: an adaption of Shiloach-Vishkin and a parallel search algorithm based on Kahan's and the Bully algorithms. The parallel search algorithm was developed by the author and provides two advantages over the Shiloach-Vishkin algorithm: for power-law

[2]Components are generally numbered 0 ... N-1, and a numerical comparison can be used to pick the winner.

graphs with a large number of components it is considerably faster and it is considerably simpler to implement. The simplicity allowed adaptations for both traditional message passing and Cray SHMEM to be developed side by side.

The algorithm consists of three phases, similar to Kahan's algorithm. Unlike Kahan's, however, hot-spots are minimized by replication and communication is well controlled, even for unbalanced graphs.

(1) Parallel searches are used to mark each vertex in the graph as belonging to a component. Collisions are stored in an ordered list of collisions. Rather than starting a parallel search at each vertex simultaneously, each process starts a single parallel search and only starts a new parallel search when the first has completed and there is no work to be done completing parallel searches started by remote processes.

(2) The individual collision lists are shared between all processes in a global all-to-all communication and a table mapping all component names used during the first step to their final component name is then constructed. Unlike Kahan's algorithm, in which there will always be $|V|$ components started, our algorithm limits the number of "false" components by limiting the number of simultaneous parallel searches.[3]

(3) Each process iterates through the vertices local to the process and updates the component name associated with the vertex based on the table generated in step 2. The table look-up is currently implemented using an STL map, and the updates are completely independent of both the graph structure and vertices on remote processes. The step is completely independent from the actions of other processes and no communication takes place during this phase. As long as the vertices in the graph are evenly distributed, this step will also load balance quite well.

Unlike the multi-threaded implementation of Kahan's algorithm, the Parallel BGL's Parallel Search algorithm does not have an issue with communication hot-spots from the collision data due to the independent, distributed nature of the collision data structure.

[3]In a graph with exactly one component, the number of entries in the hash table may be as small as the number of processes, p.

While there may be hot-spots in the local data structure during step 1, such hot-spots are actually beneficial due to the cache-based memory hierarchies found in distributed memory platforms.

2. One-Sided Communication Properties

The Parallel BGL Parallel Search connected components algorithm discussed in Section 1.3 is used to motivate our discussion from Chapter 3 as to the use of one-sided communication paradigms. As previously discussed, the Parallel Search algorithm has been implemented utilizing both the Parallel BGL process group abstraction, which provides a BSP-style communication infrastructure over MPI point-to-point communication, and over Cray SHMEM. As Cray SHMEM does not provide collectives, step 2 utilizes MPI collective routines even in the Cray SHMEM implementation.

2.1. Data-dependent Communication. Communication in step 1 of the Parallel Search algorithm is solely dependent upon the structure of the graph. When a parallel search encounters an edge to a remote vertex, communication is initiated. While messages are explicitly bundled when the Parallel BGL MPI process group is used, the pairs of communicating processes are likely to change from consecutive synchronization steps, and communication patterns will appear random for interesting graphs. As discussed in Chapter 3, the irregular communication pattern of majority of the algorithm lends itself to the use of one-sided communication paradigms.

Communication in step 2 of the Parallel Search algorithm consists of a single all-to-all message. While it is possible to efficiently implement collectives on top of one-sided communication, the lack of collective routines for one-sided applications exposes a missing feature of one-sided paradigms. An naive all-to-all pattern, as would generally be implemented by application writers to cope with the short-coming in most one-sided paradigms may perform considerably worse than an optimized collective routine, tuned for the underlying network structure.

2.2. Remote Addressing. Unlike the PageRank algorithm, which depend on properties associated with individual vertices or edges (such as the current ranking in the case of PageRank), connected components relies only on the graph structure (vertex and edge lists) and internal data structures of the algorithm's choice during execution. The Parallel Search algorithm updates a vector-based property map of current component assignments in step 1, although this structure can be changed with no loss of generality or impact to user applications.

Limiting remote addressing to a data structure internal to the algorithm greatly simplifies the addressing problem discussed in Chapter 3, Section 2.1. In the case of Cray SHMEM, the temporary component map is allocated from the symmetric heap and is indexed based on the vertex's local index. An edge to a remote process includes enough information to resolve the peer process identifier and the local vertex number on that process, allowing local resolution of the remote address.

2.3. Read-Modify-Write Atomic. Care must be taken when updating the component membership of a vertex, to solve the obvious race condition of multiple searches simultaneously attempting to update the same vertex. In the message passing implementation, messages are handled serially during a synchronization phase in which the process is not directly updating the vertex. A one-sided implementation must provide an atomic read-modify-write primitive in order to implement the algorithm.

The simplest and most efficient implementation of the algorithm utilizes a compare-and-swap primitive for all component membership updates. The vertex is assumed have an "invalid" component assignment, which was assigned during initialization. If the compare and swap succeeds, then the vertex had not yet been assigned to a component. If the operation failed, the vertex belongs to another component, and the compare and swap operation will return the vertex's existing component. The collision can then be added into the collision table, and resolved during the second and third steps of the algorithm.

Other read-modify-write operations, such as fetch-and-add, may be used to implement the algorithm, using a second vector to act as lock locations for the component value. The

implementation suffers from a much higher communication cost than the single round trip of the compare-and-swap implementation. The algorithm requires three round trip operations, in addition to the put operation if the component has not been updated. An additional round-trip for a get to find the initial state of the component assignment may be added before the lock if it is likely that the vertex has already been assigned a component.

3. One-Sided Algorithm Implementation

Following the previous discussion of the requirements of the Parallel Search connected components algorithm on one-sided communication paradigms, this section discusses an implementation of the algorithm for Cray SHMEM, as well as a discussion as to why the MPI one-sided interface is inadequate for implementing the algorithm. Performance of the Cray SHMEM implementation relative to the message-passing based implementation is also presented.

3.1. Cray SHMEM. The Parallel Search connected components algorithm presented few challenges when implemented in Cray SHMEM. The data-dependent communication patterns combined with straight-forward remote addressing simplify communication. The communication operation can be expressed as a word-sized compare-and-swap operation, one of the primitives available with all versions of Cray SHMEM. Finally, the light-weight synchronization requirements of Cray SHMEM ensures that step 1 may be implemented without any global synchronization primitives.

To limit code changes between message passing and SHMEM implementations of the Parallel Search algorithm, a SHMEM-based property map was implemented as part of the algorithm development. The SHMEM property map supports local property map put/get operations, similar to other property maps. Remote put/get operations are implemented in terms of Cray SHMEM operations and take place immediately. This results in a slight loss in semantics, as the put is not "resolving" as they are for other property maps; the put is a direct write, with no opportunity to resolve conflicts. The SHMEM property map also exposes a `start` method, which returns the start of the data array stored in the property map. The data array is allocated in the SHMEM symmetric heap, meaning that

all processes will return the same pointer from start. The start method allows algorithms to directly manipulate the data stored in remote property map instances, as is done in the Parallel Search algorithm.

```
q.push(starting_vertex);
while (!q.empty()) {
  vertex_descriptor v = q.front();
  q.pop();
  BGL_FORALL_ADJ_T(v, peer, g, Graph) {
    component_value_type my_component = get(c, v);
    component_value_type their_component = max_component;
    process_id_type owner = get(owner, peer);

    shmem_int_compare_and_swap(c.start() + local(peer),
                               my_component,
                               their_component,
                               owner);
    if (their_component != max_component) {
      collisions.add(my_component, their_component);
    } else if (id == owner) {
      // if it's local, start pushing its value early. Can't
      // do this for remote processes, because of the cost of
      // multiple−writer queues.
      q.push(peer);
    }
  }
  if (q.empty()) {
    q.push(next_vertex());
  }
}
```

FIGURE 1. Parallel Search algorithm step 1 using Cray SHMEM.

Step 1 of the algorithm, shown in Figure 1, involves the majority of the communication and takes the majority of the run-time. The algorithm differs from the message passing version in that when updating a remote vertex, that vertex is not added to the remote process's work queue. Instead, vertexes are processed in order as the algorithm iterates through the vertex list. Multiple writer queues or lists are extremely challenging to implement for Cray SHMEM. The challenges are similar to those found in lock-less shared-memory data structures, which are frequently very inefficient [**54**]. It is far simpler and likely far less

expensive to handle the potential increase in entries in the component collision table than it is to implement a multiple-writer SHMEM queue.

Because the NIC is atomically modifying the component vector with updates even as the local processor is locally updating the component vector, local operations must also use atomic memory operations. SHMEM requires that SHMEM atomic primitives be used, rather than using the processor's built-in atomic primitives. The SHMEM-based synchronization is necessary due to the loose synchronization between the processor and NIC, which are unlikely to be solved in the near future. The extra cost of local synchronization is unfortunate but necessary for correctness in our unsynchronized model.

They message passing implementation of step 1 is forced to synchronize frequently (generally whenever the local work queue is empty) to exchange communication messages. This is an unfortunate side-effect of the data-driven communication patterns, as messages are only guaranteed to be delivered by the message-passing process group at BSP-like synchronization steps. Because communication in the SHMEM implementation is via atomic operations, the communication operation has been committed to remote memory upon return of the SHMEM call, removing the need for any synchronization during step 1.

Step 2 of the Parallel Search algorithm requires an all-to-all collective communication call to start the step, and then requires no further communication. Each process duplicates the work of resolving the conflicts table, rather than more costly algorithms which trade duplicate computation (which is relatively cheap) for more communication (which is relatively expensive). The collective all-to-all does expose a significant short-coming in Cray SHMEM interface, the lack of collective operations other than a simple barrier. While efficient collective routines can certainly be implemented over SHMEM, it is unfortunate that the user is forced to handle such implementations himself. Collective performance requires careful attention to network structure and often times involves counter-intuitive performance trade-offs. Fortunately, the Cray XT series of machines used during development and benchmarking of the algorithm allows SHMEM and MPI to be used within the same process with no loss in performance for either interface. Therefore, the MPI

MPI_ALL_TO_ALL function was used to start step 2. As all-to-all has pseudo-barrier semantics[4], the call is used not only to transfer collision data between peers, but also to ensure that all processes have completed step 1 before any begin hash table resolution in step 2.

Step 3 of the algorithm is entirely local and therefore is unchanged between message passing and SHMEM implementations. While the authors do not believe it is necessary, it may be advantageous to synchronize the exit from the Parallel Search algorithm between the participating processes. In this case, a barrier at the end of step 3 would be necessary. Cray SHMEM provides a barrier operation, which would be sufficient for this use.

3.2. MPI One-Sided. MPI presents a number of insurmountable barriers for implementing the Parallel Search connected components algorithm. In particular, the interface lacks a read-modify-write atomic operation and the buffer access restrictions of the standard impose a heavy synchronization cost. The lack of an appropriate atomic operation prohibits an implementation of the Parallel Search algorithm, although the heavy synchronization cost would render an implementation impractical.

The MPI one-sided interface provides an atomic update call, MPI_ACCUMULATE, which supports a number of atomic arithmetic operations on both integer and floating point datatypes. The operation does not, however, return either the previous or updated value. Further, an MPI_ACCUMULATE or MPI_PUT followed by an MPI_GET to the same memory address is prohibited by the standard, eliminating the use of MPI_LOCK/MPI_UNLOCK for emulating the atomic operation.

Assuming the MPI_ACCUMULATE function returned the value in true read-modify-write fashion, the synchronization and buffer access rules of the MPI standard would still severely limit the performance of a connected components algorithm. The algorithm depends on immediately determining the remote vertex's component assignment. Assuming a modified MPI_ACCUMULATE exhibits the same behavior as MPI_GET in terms of data availability, the algorithm's implementation would be forced to using lock/unlock synchronization.

[4]MPI does not actually guarantee full barrier semantics for all-to-all collective calls. However, no process may leave the all-to-all until it has received every other process's data. This data dependency provides the limited barrier semantics we require.

As discussed in Section 3.2, lock/unlock synchronization requires at least one round-trip communication. A true one-sided implementation would require three round-trip communications (one to acquire the lock, one for the communication operation, and finally a third to release the lock).

3.3. Performance Results. Results of the Parallel BGL's Parallel Search connected components algorithm are presented using both MPI send/receive semantics and Cray SHMEM. As discussed in Section 3.2, there is not an implementation of the algorithm using the MPI one-sided interface. Both the MPI send/receive and Cray SHMEM implementations are compared on the Red Storm machine.

3.3.1. *Test Environment.* Tests were performed on the Red Storm machine at Sandia National Laboratories. Red Storm is a Massively Parallel Processor machine, which later became the basis of the Cray XT platform line. In the configuration used, the platform includes 6,720 dual-core 2.4 GHz AMD Opteron processors with 4 GB of RAM and 6,240 quad-core 2.2 GHz AMD Opteron processors with 8 GB of RAM. Each node contains a single processor, and nodes are connected by a custom interconnect wired in a semi-torus topology (two of the three directions are wired in a torus, the other does not wrap around to allow for red/black switching). Each node is connected to the network via a custom SeaStar network interface adapter, capable of providing 4.78 μs latency and 9.6 GB/s bandwidth.

UNICOS/lc 2.0.61.1 and the Catamount light-weight kernel were running on the system during testing. Cray's XT MPI library, based on MPICH2, and the Cray-provided SHMEM libraries were used for message passing and SHMEM, respectively.

3.3.2. *Test Graph Structures.* The structure of a graph can greatly influence the performance of an algorithm, as can be seen in Section 3.3.3. Three graphs with very different structures are used to compare the two connected component algorithm implementations. The first is an Erdös-Rényi graph, a uniformly random graph. The other two are based upon the R-MAT graph generator, which is capable of generating high-order vertices.

The Erdös-Rényi [**27**] graph model generates a random graph in which each pair of vertices have equal probability of being connected by an edge. The format does not model

graphs which tend to occur in real life, but is extremely useful in proving traits about both graph properties and graph algorithms. Because the out-degree of each vertex is probabilistically uniform, load balancing of Erdös-Rényi graphs are much easier than other graph structures. The probability p of an edge between any two vertices in the graph used for testing is .0000001. The number is low so that the number of edges does not explode as the number of vertices grows.

The R-MAT (Recursive MATrix) [**24**] graph generator generates random graphs which are parametrized to produce a variety of graph structures. Parameters can be chosen which mimic a variety of real-world data sets. Four parameters, generally referred to as a, b, c, and d, are used to generate the graph. The generator is recursive, dividing the vertices of a graph into four partitions, and choosing a partition based on the probabilities a, b, c, and d, repeating until a vertex is chosen, then the process is repeated to pick its pair. The procedure then is repeated until the proper number of edges are generated. Duplicate edges may be generated during the procedure, which are thrown out and a new edge generated.

The parameters a, b, c, and d determine the graph structure. Using $a = 0.25$, $b = 0.25$, $c = 0.25$, and $d = 0.25$ will generate an Erdös-Rényi graph. Putting more weight on one of the quadrants generates an inverse power-law distribution. Two sets of parameters are used in the tests:

nice: *Nice* graphs use the parameters $a = 0.45$, $b = 0.15$, $c = 0.15$, and $d = 0.25$. The graph generated features two communities at each level of recursion in quadrants a and d. The maximum vertex degree is roughly 1,000 in graphs with 250,000 vertices. While more difficult to load balance than Erdös-Rényi, the load balancing issues are still surmountable with little work.

nasty: *Nasty* graphs use the parameters $a = 0.57$, $b = 0.19$, $c = 0.19$, and $d = 0.05$. Due to the heavy weighting of quadrant a, the maximum degree for the 250,000 vertices is closer to 200,000. The load balancing issues with a *nasty* graph are much more difficult than with both Erdös-Rényi and R-MAT graphs with *nice* parameters.

During graph generation, the vertex ids generated by the R-MAT graphs are permuted by a random, uniform permutation vector. Without the permutation, the vertices for each quadrant would tend to be placed on the same set of processors. With the permutation vector, it is likely that all nodes will have an equal number of vertices from each quadrant.

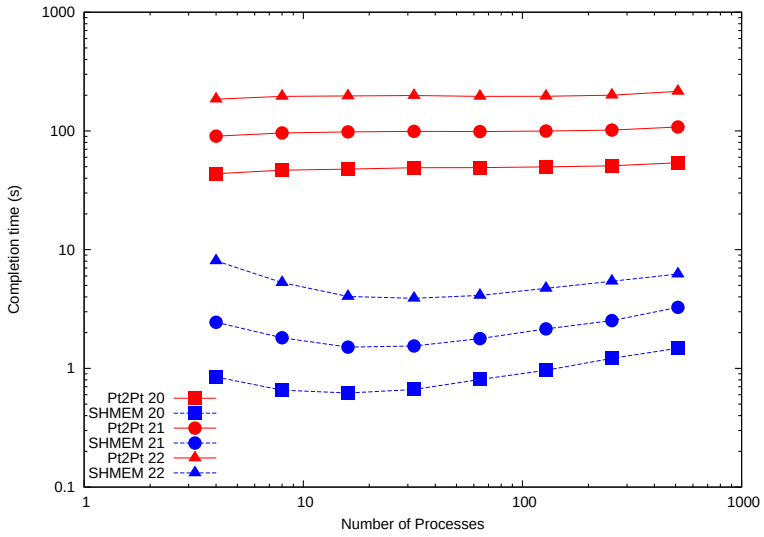


FIGURE 2. Connected Components completion time, using an Erdös-Rényi graph with edge probability .0000001.

3.3.3. *Analysis.* As shown in Figure 2, the Cray SHMEM implementation performs significantly better than the MPI implementation of the connected components algorithm for Erdös-Rényi graphs. There is little hot-spotting in the algorithm which would cause contention at a given node. There are a small number of equally large components in the graph, which means that there will be small messages sent to a large number of nodes at every synchronization point in the algorithm.

Unlike the Erdös-Rényi graphs, the *nice* R-MAT graphs cause significant performance degradation for the Cray SHMEM implementation, as seen in Figure 3. The *nice* R-MAT graph has a number of components, mostly equal in size. There are a number of large components, which limits the number of messages that must be sent, as a component that cross a processor boundary requires only one message. The high cost of local updates in the SHMEM case, which requires a compare and swap, can not be overcome by the lower communication and synchronization cost.

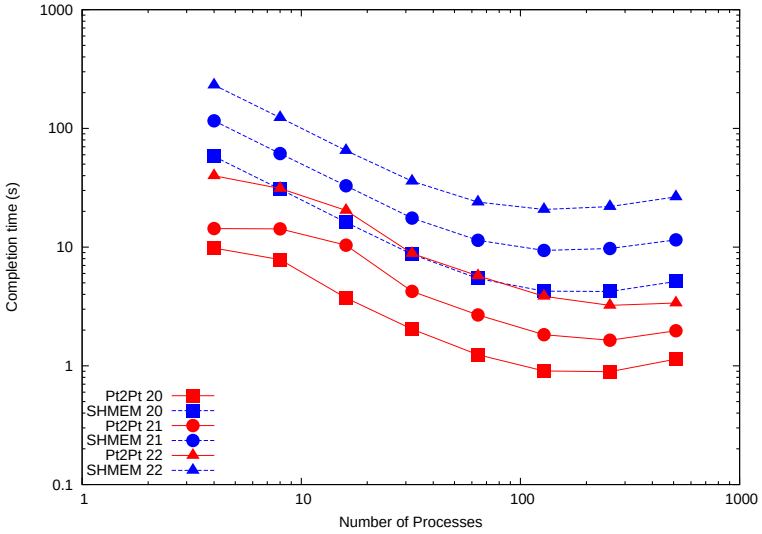


FIGURE 3. Connected Components completion time, using the "nice" R-MAT parameters, and average edge out-degree of 8.

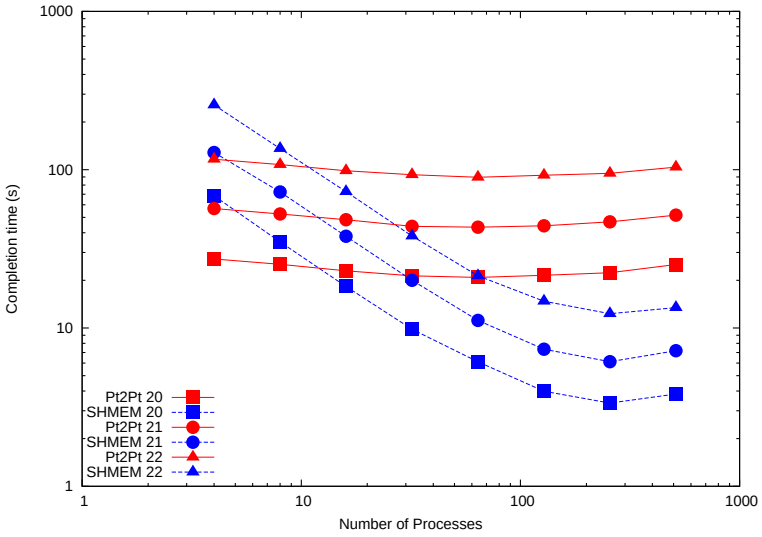


FIGURE 4. Connected Components completion time, using the "nasty" R-MAT parameters, and average edge out-degree of 8.

The *nasty* R-MAT graph results, presented in Figure 4, demonstrate the ability of one-sided communication paradigms to overcome the load balancing issues inherent in graphs with *nasty* parameters. The *nasty* graph has a significant (1,000–2,000) number of components. Generally, one component will encompass the majority of the vertices, with the remainder of components having a small number of vertices (1–10). The small components

present a problem for the message passing implementation, as the synchronization when completing the small components present a high performance cost.

4. Conclusions

The Parallel Search Connected Components algorithm is currently the most scalable connected components algorithm in the Parallel BGL. The algorithm provides a good match to one-sided communication, based on the parameters discussed in Chapter 3. As discussed in Section 3, Cray SHMEM supports the connected components algorithm with much simpler code than the MPI implementation, and as shown in Section 3.3 the performance of the algorithm is better than the message passing implementation. On the other hand, the MPI one-sided interface presents insurmountable difficulties for implementation.

CHAPTER 5

Case Study: PageRank

PageRank is the algorithm behind Google's search engine, measuring the "importance" of a web page based on the number and importance of other pages linking to it. [**17**] Various characteristics (web pages, physical connections, etc.) of the Internet are often modeled as a large graph algorithm, and PageRank can be implemented using such a data representation. PageRank and slight variations on the original algorithm have found applicability outside of search engines. [**94, 12**] In addition to graph representations, PageRank has been successfully implemented using a number of different programming paradigms, including Map-Reduce and as a traditional sparse linear algebra problem. [**76**]

1. PageRank Algorithm

The general theory behind PageRank is that "important" pages are linked to by other "important" pages. Initially, each vertex in the graph has the same importance rank, generally 1.0. The rank value is traditionally a real number between 0.0 and 1.0, which can be viewed as the probability a given vertex would be found in a random walk of the graph. Unlike connected components, which is only applicable to undirected graphs, PageRank is only applicable to directed graphs.

PageRank consists of a (generally bounded) number of iterations during which the rank values flow along out-edges in the graph. For each iteration, a vertex v is updated according to Equation 1.[1]

$$PR(v_i) = \frac{1-d}{|V|} + \sum_{v_j \in ADJ(v_i)} \frac{PR(v_j)}{OUT(v_j)} \tag{1}$$

[1]The literature is inconsistent as to whether the $1-d$ term is divided by $|V|$. In either case, the computational complexity and communication patterns are identical.

$OUT(v_J)$ is the total number of out-edges for v_j, so that no new "rank value" is created during the algorithm step. $ADJ(v_i)$ is the set of vertices adjacent to v_i. In a double-buffered scheme, $PR(v_j)$ is generally the page rank of v_j as determined by the previous iteration of the algorithm.

Generally, the algorithm utilizes two rank values for each vertex in a double buffering scheme. A serial implementation of the algorithm is shown in Figure 1. While not shown in the code, the rank value should be re-normalized to the 0.0 to 1.0 range at the completion of each step. Completion of the algorithm is determined either by a fixed number of steps or when the maximum change in any vertex's rank between two iterations falls below a specified threshold.

```
void page_rank_step(const Graph& g, RankMap from_rank, RankMap to_rank,
                    double damping)
{
  // update initial value with damping factor
  BGL_FORALL_VERTICES_T(v, g, Graph) put(to_rank, v, rank_type(1 − damping));

  // ''push'' rank value to adjacent vertices
  BGL_FORALL_VERTICES_T(u, g, Graph) {
    rank_type u_rank_out = damping ∗ get(from_rank, u) / out_degree(u, g);
    BGL_FORALL_ADJ_T(u, v, g, Graph)
      put(to_rank, v, get(to_rank, v) + u_rank_out);
  }
}
```

FIGURE 1. Pseudo-code for a single PageRank iteration step using a push model.

2. One-Sided Communication Properties

2.1. Data-dependent Communication. Unlike the connected components communication pattern discussed in Chapter 4, PageRank's communication pattern is deterministic. At each step in the algorithm, data must be transferred for each edge in the graph, as the rank value from the source vertex is pushed to the target vertex of the edge. In a distributed implementation, assuming the graph does not change during the algorithm (which is generally the case), the total amount of data to be sent in a given iteration from

a process to any other process can be determined at any time, including during the initial "setup" phase of the algorithm.

The communication pattern of PageRank is largely dependent upon the structure of the underlying graph. Power-law graphs, in which a small percentage of the vertices have high connectivity and the majority of vertices have a low number of edges, are likely to result in a small number of processors participating in the majority of the communication. Erdös-Rényi graphs, on the other hand, are likely to involve communication with a uniformly high number of remote processes, with more balanced communication sizes.

Although the communication pattern is deterministic, making two-sided communication easier to organize than with connected components, the high number of remote peers still presents a challenge. For large problem sizes distributed among a high number of peers, a simple model which posts a receive from each peer may be insufficient due to performance issues associated with large posted receive counts. [**89**]

2.2. Remote Addressing. Similar to connected components in Chapter 4, care must be taken in the storage of rank value data. If the rank data is stored in the vertex data structure, the graph is limited to those storage structures in which the address of the remote vertex can easily be computed from the origin process. This generally eliminates list-based structures, as encoding a node and address can be space prohibitive. An array based structure, requiring only a node id and local index be encoded, is much less prohibitive. If external storage, like Parallel BGL's property map, is used, similar restrictions are placed on the property map data structure. Dynamic graphs with vertex rank computations stored from previous algorithm runs further complicate the issue, as dynamic graphs are generally stored in list-based representations to allow easier modification of the graph.

2.3. Floating-point Atomic Operation. The push-based PageRank algorithm requires the rank of v_i be atomically added to the rank of v_j. Unlike connected components, where the algorithm must know the old target component value, the PageRank algorithm has no need for the original rank value of v_j. A remote atomic addition primitive or locking mechanism is required to implement the algorithm. Unfortunately, while remote atomic

integer addition is a common one-sided primitive, PageRank requires floating point addition operations. In order to be truly one-sided, the library must avoid interrupting the target's host process, so a floating point atomic would require a floating point unit on the network interface card. The update operation may also be implemented using a synchronized update, either a lock followed by a get/put combination or using a get followed by a compare-and-swap (Figure 2). However, such an operation requires a cost-prohibitive minimum of two round-trips to the remote processor's memory.

```
double start, ret, tmp;
shmem_double_get(&ret,
                 to_rank.start() + local(v),
                 1, get(owner, v));
do {
  start = ret;
  tmp = start + shmem_get(from_rank, u);
  ret = shmem_cswap((long int*) (to_rank.start() + local(v)),
                    *((long int*) &start),
                    *((long int*) &tmp),
                    get(owner, v));
} while (ret != start);
```

FIGURE 2. Implementation of remote atomic floating point via compare-and-swap

3. One-Sided Algorithm Implementation

The graph-based implementation of PageRank can be implemented using both Cray SHMEM and MPI one-sided. Details of the implementation are discussed, particularly with a focus on the implementation's short-comings relative to the taxonomy discussed in Chapter 3. Performance comparisons between the three implementations are also provided.

3.1. Cray SHMEM. Cray SHMEM has always been designed to closely match the underlying hardware. As floating-point unites have only recently shrunk to the point where they can be cost-effectively implemented on a NIC, Cray SHMEM does not provide an atomic remote floating-point addition. Therefore, the implementation options for Cray SHMEM are limited to a push model similar to Figure 3, or a pull model (Figure 4). The cost

of two serialized round-trip network communications is prohibitive, with performance on the test graph sizes an order of magnitude worse than the bidirectional pull implementation. The limitation of Cray SHMEM to integer math is a combination of software implementation choices and hardware deficiencies. The SHMEM paradigm would easily support a floating point remote add if such an operation was supported without interrupting the host.

```
BGL_FORALL_VERTICES_T(u, g, Graph) {
  put(from_rank, u, (damping * get(from_rank, u) / out_degree(u, g)));
  BGL_FORALL_ADJ_T(u, v, g, Graph) {
    double start, ret, tmp;;
    shmem_double_get(&ret,
                     to_rank.start() + local(v),
                     1, get(owner, v));
    do {
      start = ret;
      tmp = start + get(from_rank, u);
      ret = shmem_cswap((long int*) (to_rank.start() + local(v)),
                        *((long int*) &start),
                        *((long int*) &tmp),
                        get(owner, v));
    } while (ret != start);
  }
}
shmem_barrier_all();

BGL_FORALL_VERTICES_T(v, g, Graph) put(from_rank, v, rank_type(1 - damping));
```

FIGURE 3. Cray SHMEM implementation of the PageRank update step, using a "push" model, which does not require a bi-directional graph.

3.2. MPI One-Sided. MPI one-sided provides a remote atomic operation via the MPI_ACCUMULATE function. MPI_ACCUMULATE is capable of operating on any pre-defined MPI datatype, including floating point numbers. Operations include addition, subtraction, multiplication, minimum, and maximum. Like all MPI one-sided communication calls, MPI_ACCUMULATE is non-blocking. Unlike MPI_PUT and MPI_GET's highly restrictive prohibition on multiple updates to a single target address in a synchronization period, multiple MPI_ACCUMULATE calls may update the same target address in the same synchronization phase.

```
BGL_FORALL_VERTICES_T(v, g, Graph) {
  put(from_rank, v, get(from_rank, v) / out_degree(v, g));
}
shmem_barrier_all();
BGL_FORALL_VERTICES_T(v, g, Graph) {
  rank_type rank(0);
  BGL_FORALL_INEDGES_T(v, e, g, Graph) {
    double ret;
    shmem_double_get(&ret,
                     from_rank.start() + local(source(e, g)),
                     1, get(owner, source(e, g)));
    rank += ret;
  }
  put(to_rank, v, (1 − damping) + damping ∗ rank);
}
shmem_barrier_all();
```

FIGURE 4. Cray SHMEM implementation of the PageRank update step, using a bi-directional graph and the "pull" algorithm.

Assuming that the graph is well partitioned (or even randomly partitioned), it can be assumed that communication with the majority of peers in the parallel application will be required in any single step of the algorithm. Therefore, MPI_FENCE synchronization, which is collective across all communicating processes, is used for synchronization. An ideal implementation using the MPI one-sided implementation is shown in Figure 5.

```
BGL_FORALL_VERTICES_T(u, g, Graph) {
  put(from_rank, u, (damping ∗ get(from_rank, u) / out_degree(u, g)));
  BGL_FORALL_ADJ_T(u, v, g, Graph) {
    MPI_Accumulate(&(from_rank[u]),
                   1, MPI_DOUBLE,
                   get(owner, v), local(v),
                   1, MPI_DOUBLE, MPI_SUM, to_win);
  }
}
MPI_Win_fence(0, to_win);

// Set new rank maps for map which will be the target in the next step
BGL_FORALL_VERTICES_T(v, g, Graph) put(from_rank, v, rank_type(1 − damping));
```

FIGURE 5. MPI one-sided implementation of the PageRank update step

Due to inefficiencies in current MPI one-sided implementations, the straight-forward algorithm does not offer performance comparable to the point-to-point PageRank implementation. MPI one-sided, due to its use of complicated MPI datatypes [2], must send a large amount of data, 40 bytes in the case of Open MPI, and buffering all messages between fence operations exhausts memory resources. In order to prevent message or memory exhaustion, a call to MPI_FENCE is made every 100,000 calls to MPI_ACCUMULATE (Figure 6). In practice, a fence every 100,000 to 500,000 appears to prevent advanced MPI implementations from exhausting resource, while not greatly impacting performance due to extra synchronization.

```
BGL_FORALL_VERTICES_T(u, g, Graph) {
  put(from_rank, u, (damping * get(from_rank, u) / out_degree(u, g)));
  BGL_FORALL_ADJ_T(u, v, g, Graph) {
    MPI_Accumulate(&(from_rank[u]),
                   1, MPI_DOUBLE,
                   get(owner, v), local(v),
                   1, MPI_DOUBLE, MPI_SUM, to_win);
    count++;
    if ((count % num_per_fence) == 0) {
      MPI_Win_fence(0, to_win);
      fcount++;
    }
  }
}
for (int i = fcount ; i < extra_fences ; ++i) {
  MPI_Win_fence(0, to_win);
}
MPI_Win_fence(0, to_win);

// Set new rank maps for map which will be the target in the next step
BGL_FORALL_VERTICES_T(v, g, Graph) put(from_rank, v, rank_type(1 - damping));
```

FIGURE 6. MPI one-sided implementation of the PageRank update step, with added synchronization

3.3. Performance Results. Comparisons of the three implementations of PageRank are provided in this section. The tests were conducted on the Red Storm platform, which

[2]User defined datatypes of arbitrary complexity are allowed, as long as they are composed of only one MPI pre-defined datatype.

was described in Chapter 4, Section 3.3. The same graph structures previously presented are used for the PageRank comparisons.

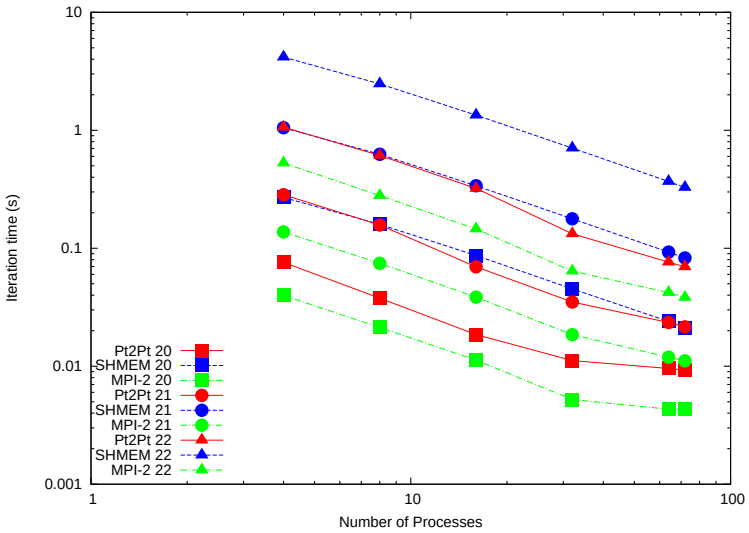


FIGURE 7. PageRank completion time, using an Erdös-Rényi graph with edge probability .0000001.

3.3.1. *Analysis.* The performance issues due to the blocking get implementation of Cray SHMEM described in Section 3.1 can clearly be seen in the performance results for all three graph types. It was initially somewhat surprising that the "nasty" R-MAT parameters in Figure 9 didn't produce a different scaling curve for Cray SHMEM than that found in the "nice" R-MAT parameters (Figure 8) or the Erdös-Rényi graph (Figure 7). The vertex permutation step prevents a small number of nodes from having the majority of the high out-degree vertices. The average out-degree of both the "nasty" and "nice" graphs is 8, even if the distribution of those edges varies widely. The vertex permutation step means that the average number of edges on a given process for either R-MAT graph is similar, and the number of blocking get calls is therefore also similar for each processes.

MPI one-sided and the MPI message passing implementations have similar performance characteristics for all three graph structures. The more partitionable Erdös-Rényi and "nice" R-MAT graphs show a performance advantage for the one-sided implementations, although that disappears as the graph becomes more challenging with the "nasty" graphs. The one-sided implementation within Open MPI is much more capable of handling the

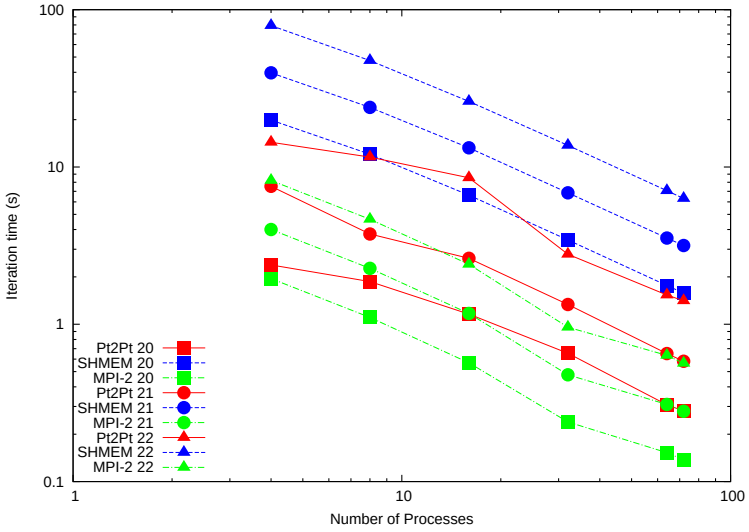


FIGURE 8. PageRank completion time, using the "nice" R-MAT parameters, and average edge out-degree of 8.

usage pattern found in PageRank than other MPI implementations[3], although it still has a difficult time dealing with the high number of outstanding sends.

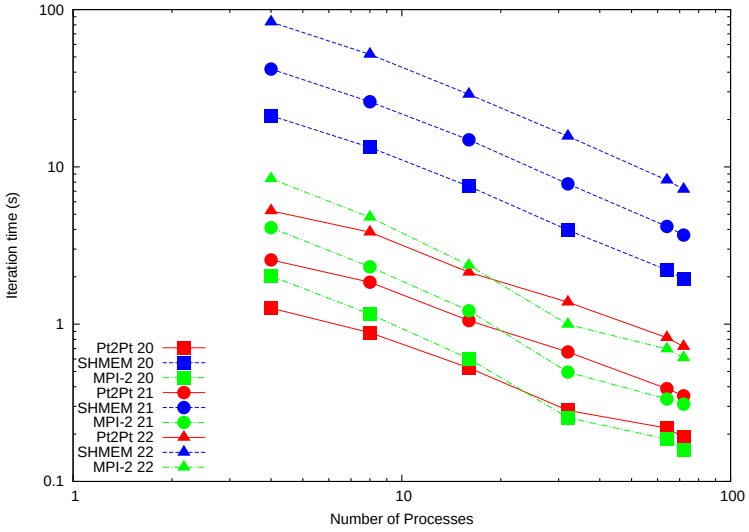


FIGURE 9. PageRank completion time, using the "nasty" R-MAT parameters, and average edge out-degree of 8.

[3]Due to limitations with the Cray MPI implementation, there can only be 2,000 outstanding requests in a given Epoch, hardly practical for our usage.

4. Conclusions

In the previous chapter, a weakness in the MPI one-sided interface proved too substantial for an implementation of the connected components algorithm, but Cray SHMEM provided an ideal interface for the implementation. In this chapter, the strength of MPI's non-blocking atomic operation and explicit synchronization is shown. At the same time, the blocking get interface and lack of floating-point atomic operations poses both implementation and performance issues for Cray SHMEM.

The results suggest that non-blocking a communication interface is important for situations in which a high degree of natural parallelism exists. More importantly, it verifies our belief from Chapter 3 that the prevailing believe that "universal" atomic operations, while equivalent semantically, do not have the same performance equivalence in one-sided as in local memory operations. The atomic addition implemented using compare-and-swap performed so poorly that results for reasonable sized problems could not be generated. At the same time, the MPI non-blocking atomic operation was able to perform at a level better than or comparable to the message-passing implementation. Due to this result, one-sided interfaces require a much larger set of atomic primitives than a traditional memory system.

CHAPTER 6

Case Study: HPCCG

The previous two case studies examined applications which are centered around graph-based informatics. More traditional physics-based applications have different communication patterns and requirements than the informatics codes. This chapter examines a micro-application designed to exhibit many of the performance characteristics of large-scale physics codes. Although the performance difference between the various communication paradigms is practically zero, synchronization issues are problematic for one-sided interfaces which provide implicit synchronization.

1. HPCCG Micro-App

The HPCCG Micro-App, part of the Mantevo benchmark suite, is designed to be the "best approximation to an unstructured implicit finite element or finite volume application in 800 lines or fewer". [**77**] HPCCG provides a small application with many of the same computation and communication properties found in traditional large scale physics applications. Unlike graph-based informatics codes, which communicate with a large number of peers, physics codes generally talk to a small number of peers, and the list of peers does not change significantly during the lifespan of the application. The message passing implementation of the HPCCG ghost-cell exchange is shown in Figure 1.

Finite element physics applications generally break down a physical 1, 2, or 3 dimensional space into a number of discrete "points", then run a calculation simulating some physical event based on the values at each point. Updates are generally based not only on the point, but it's closest neighbors in the space. When the space is partitioned to run on multiple independent memory spaces, these neighbors must be shared at the conclusion of each iteration, hence the stable, small number of communication peers.

```
MPI_Request * request = new MPI_Request[num_neighbors];

// Externals are at end of locals
double *x_external = (double *) x + local_nrow;

for (i = 0; i < num_neighbors; i++) {
  int n_recv = recv_length[i];
  MPI_Irecv(x_external, n_recv, MPI_DOUBLE, neighbors[i], MPI_MY_TAG,
            MPI_COMM_WORLD, request+i);
  x_external += n_recv;
}

// Fill up send buffer
for (i=0; i<total_to_be_sent; i++) send_buffer[i] = x[elements_to_send[i]];

for (i = 0; i < num_neighbors; i++) {
  int n_send = send_length[i];
  MPI_Send(send_buffer, n_send, MPI_DOUBLE, neighbors[i], MPI_MY_TAG,
           MPI_COMM_WORLD);
  send_buffer += n_send;
}

MPI_Waitall(num_neighbors, request, MPI_STATUSES_IGNORE);

delete [] request;
```

FIGURE 1. Message passing implementation of the HPCCG ghost-cell exchange.

Message sizes are driven by the size of the shared face, the set of points that adjoin another process's data points. In a well partitioned problem, however, a large shared face implies a larger amount of data without any shared neighbors. This low surface to volume ratio implies that as the face size grows, the amount of computation in each iteration grows much faster. Thus, the communication phase will generally be dominated by the computation phase of physics codes which are limited by memory size more than by processing power.

Not all codes have the same straight-forward distribution as the HPCCG micro-application. The physics being modeled may require a much higher ratio of surface to volume, particularly in cases where high energies are involved. In these cases, the communication cost may

quickly dominate the computation time. We believe that the results presented in this chapter apply to such codes, although the implicit synchronization problem discussed becomes much more severe in these cases.

2. One-Sided Communication Properties

The ghost cell exchange of the HPCCG application (and similar physics applications) are based on a partitioning of the data and not the data itself. Unlike the previous two case studies, communication is to a small number of peers, determined based on the need for "ghost cells", or local copies of remote data, for the boundary values which result from the partitioning. For static decompositions of the data, the only type supported in HPCCG, the communication pattern is set at the start of the problem.

Unlike the previous two case studies, the HPCCG communication is based solely on a *put* primitive, with no atomic requirements. The number of operations in each iteration of the algorithm is small enough that the effect of non-blocking versus blocking interface is also minimized. The high computation to communication ratio also helps cover the slight cost of a blocking implementation, meaning that the small performance increase may not be worth the added complexity of some non-blocking interfaces.

HPCCG's synchronization requirements also differ from both the connected components and PageRank applications. Connected Components required no synchronization during the core of the algorithm and PageRank required global synchronization to complete each iteration of the algorithm. In the case of HPCCG, the next iteration can begin as soon as all ghost cell data has been received. Since the number of peers involved in the ghost cell exchange is much smaller than the number of processes in the job, a global synchronization mechanism may be overkill. Instead, synchronization based on a small number of active processes is needed.

3. One-Sided Algorithm Implementation

Figure 1 demonstrates the message passing implementation of the ghost cell exchange. Once the initial sparse matrix has been created, the ghost cell exchange is one of two steps

in the communication phase. The other step is an all-reduce of the residual to check for completion. The reduction collective is not implemented in one-sided, similar to previous case studies. Unlike the other case studies, the message passing implementation does not rely on a large underlying library to hide a complex synchronization and buffering scheme.

3.1. Cray SHMEM. The Cray SHMEM implementation, shown in Figure 2, utilizes non-blocking put calls to transfer the ghost cell buffer directly into the array on the receiving process. On platforms with high message rates for one-sided implementations, the copy into send_buffer and the single shmem_double_put_nb to each peer could be replaced with multiple calls to shmem_double_put_nb. On the receive side, the sparse matrix format stores the remote data points contiguously at the end of the array, meaning there is no need to copy data before the computation phase.

```
for (i=0; i<total_to_be_sent; i++) send_buffer[i] = x[elements_to_send[i]];

for (i = 0; i < num_neighbors; i++) {
  int n_send = send_length[i];

  shmem_double_put_nb((double*)x + peer_offsets[i], send_buffer,
                      n_send, neighbors[i], NULL);
  send_buffer += n_send;
}

shmem_barrier_all();
```

FIGURE 2. Cray SHMEM implementation of the HPCCG ghost cell exchange.

All communication must be complete before the next computation phase. Cray SHMEM's synchronization mechanism poses a significant problem for completion, as there is generally no target-side notification data has arrived. There is, however, a global fence operation in which all communication from all peers has completed before any process exits the fence. Given the high ratio of computation to communication found in HPCCG, the performance of the code with a global synchronization does not appear to be significantly lower than other implementations. A completion buffer with a local fence between the data put and

the completion put operation could be used to avoid the global synchronization, although it requires polling on a number of memory addresses. Such a design is shown in Figure 3.

```
for (i=0; i<total_to_be_sent; i++) send_buffer[i] = x[elements_to_send[i]];

for (i = 0; i < num_neighbors; i++) {
  int n_send = send_length[i];
  int tmp = 1;

  shmem_double_put_nb((double*)x + peer_offsets[i], send_buffer,
                      n_send, neighbors[i], NULL);
  shmem_fence(neighbors[i]);
  shmem_put_nb(completions[i], &tmp, 1, neighbors[i], NULL);
  send_buffer += n_send;
}

for (i = 0 ; i < num_neighbors ; i++) {
  while (completions[i] != 1) { ; }
  completions[i] = 0;
}
```

FIGURE 3. Cray SHMEM implementation of the HPCCG ghost cell exchange using polling completion.

3.2. MPI One-Sided. The MPI one-sided implementation of the HPCCG ghost-cell exchange, shown in Figure 4, avoids the completion problems of the Cray SHMEM implementation. MPI one-sided provides a generalized explicit synchronization mechanism for situations like the HPCCG application. The MPI one-sided implementation involves more network-level communication than the message passing implementation. Even in implementations which reduce network traffic at the cost of asynchronous communication patterns, a message is required for every peer in the MPI_Win_post group, and another is required for all peers in the MPI_Win_complete call. Similar to the heavy synchronization of Cray SHMEM, however, the extra messaging causes no significant impact to application performance.

Similar to the Cray SHMEM implementation, the MPI one-sided implementation can be converted to sending multiple messages and avoiding the memory copy. However, unless the message injection rate is high enough to support the transfer without buffering, there will be no advantage to avoiding the initial copies.

```
for (i=0; i<total_to_be_sent; i++) send_buffer[i] = x[elements_to_send[i]];

MPI_Win_post(wingroup, 0, win);
MPI_Win_start(wingroup, 0, win);

for (i = 0; i < num_neighbors; i++) {
  int n_send = send_length[i];

  MPI_Put(send_buffer, n_send, MPI_DOUBLE, neighbors[i], peer_offsets[i],
          n_send, MPI_DOUBLE, win);
  send_buffer += n_send;
}

MPI_Win_complete(win);
MPI_Win_wait(win);
```

FIGURE 4. MPI one-sided implementation of the HPCCG ghost cell exchange.

3.3. Performance Results. Comparisons of the three HPCCG communication implementations were performed on the Red Storm machine described in Chapter 4, Section 3.3. HPCCG utilizes weak scaling, meaning the size of the problem increases as the number of processes increases. Figure 5 presents the performance of the three implementations of the HPCCG ghost cell exchange. The flat performance graph demonstrates linear performance scaling for all three implementations. The high ratio of computation to communication means that the slight communication and synchronization overhead of both one-sided interfaces does not hinder overall application performance. Further, the unnecessary matching logic of MPI may help offset the minor performance penalty of the one-sided interfaces.

4. Conclusions

HPCCG presents an interesting case study for the one-sided paradigm in that the paradigm presents no obvious advantages over message passing. In fact, the one-sided implementations require more code to implement, are more complex due to the memory allocation limitations of one-sided implementations, and require more message traffic in all cases than a message-passing implementation. It is our belief that these limitations are all necessary to the paradigm, and not an artifact of one or more one-sided implementations. Each

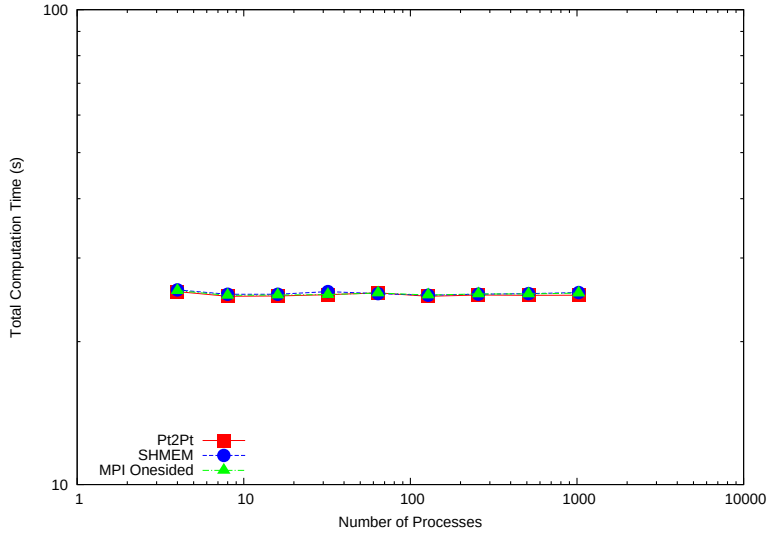


FIGURE 5. HPCCG computation time, using weak scaling with a 100x100x100 size per process.

one of these limitations are small and certainly do not make the one-sided unsuitable for implementing HPCCG and the physics codes it models.

The limitations of the one-sided paradigm presented in this Chapter, however, do suggest limits on the suitability of the one-sided paradigm for replacing message passing on heavily multi-core/multi-threaded platforms. The "heavy" synchronization and communication cost of the message passing paradigm are presented as limitations which can not be overcome on such resource limited platforms. At the same time, the one-sided paradigm requires requires a different, but costly, overhead for explicit synchronization which is naturally required. Although careful application development could likely minimize the costs associated with the explicit synchronization, it is also likely that careful application development could also avoid the costly overheads of message passing.

CHAPTER 7

MPI One-Sided Implementation

The MPI specification, with the MPI-2 standardization effort, includes an interface for one-sided communication, utilizing a rich set of synchronization primitives. Although the extensive synchronization primitives have been the source of criticism [**14**], it also ensures maximum portability, a goal of MPI. The MPI one-sided interface utilizes the concept of exposure and access epochs to define when communication can be initiated and when it must be completed. Explicit synchronization calls are used to initiate both epochs, a feature which presents a number of implementation options, even when networks support true remote memory access (RMA) operations. This chapter presents two implementations of the one-sided interface for Open MPI, both of which were developed by the author [**7**].

1. Related Work

A number of MPI implementations provide support for the MPI one-sided interface. LAM/MPI [**18**] provides an implementation layered over point-to-point, which does not support passive synchronization and performance generally does not compare well with other MPI implementations. Sun MPI [**15**] provides a high performance implementation, although it requires all processes be on the same machine and the use of MPI_ALLOC_MEM for optimal performance. The NEC SX-5 MPI implementation includes an optimized implementation utilizing the global shared memory available on the platform [**88**]. The SCI-MPICH implementation provides one-sided support using hardware reads and writes [**93**].

An implementation within MPICH using VIA is presented in [**33**]. MPICH2 [**4**] includes a one-sided implementation implemented over point-to-point and collective communication. Lock/unlock is supported, although the passive side must enter the library to make progress.

The synchronization primitives in MPICH2 are significantly optimized compared to previous MPI implementations [**87**] and influenced this work heavily. MVAPICH2 [**44**] extends the MPICH2 one-sided implementation to utilize InfiniBand's RMA support. MPI_PUT and MPI_GET communication calls translate into InfiniBand put and get operations for contiguous datatypes. MVAPICH2 has also examined using native InfiniBand for Lock/Unlock synchronization [**48**] and hardware support for atomic operations [**78**].

2. Implementation Overview

Similar to Open MPI's Point-to-point Matching Layer(PML), which allows multiple implementations of the MPI point-to-point semantics, the One-Sided Communication (OSC) framework in Open MPI allows for multiple implementations of the one-sided communication semantics. Unlike the PML, where only one component may be used for the life of the process, the OSC framework selects components per window, allowing optimizations when windows are created on a subset of processes. This allows for optimizations when processes participating in the window are on the same network, similar to Sun's shared memory optimization.

Open MPI 1.2 and later provides two implementations of the OSC framework: pt2pt and rdma. The pt2pt component is implemented entirely over the point-to-point and collective MPI functions. The original one-sided implementation in Open MPI, it is now primarily used when a network library does not expose RMA capabilities, such as Myrinet MX [**62**]. The rdma component is implemented directly over the BML/BTL interfaces and supports a variety of protocols, including active-message send/receive and true RMA. Both components share the similar synchronization designs, although the rdma component starts communication before the synchronization call to end an epoch, while the pt2pt component does not.

Sandia National Laboratories has utilized the OSC framework to implement a component utilizing the Portals interface directly, rather than through the BTL framework, allowing the use of Portal's advanced matching features. The implementation utilizes a synchronization design similar to that found in the pt2pt and rdma components. As all

three components duplicated essentially the same code, there was discussion of breaking the OSC component into two components, one for synchronization and one for communication. In the end, this idea was rejected as the synchronization routines do differ in implementation details, such as when communication callbacks occur, that could not easily be abstracted without a large performance hit.

The implementations can be divided into two parts: communication and synchronization. Section 3 details the implementation of communication routines for both the pt2pt and rdma components. Section 4 then explains the synchronization mechanisms for both components.

3. Communication

The pt2pt and rdma OSC components differ greatly in how data transfer occurs. The pt2pt component lacks many of the optimizations later introduced in the rdma component, including message buffering and eager transfers. Both components leverage existing communication frameworks within Open MPI for communication: the PML framework for pt2pt and the PML, BTL, and BML frameworks for rdma. Both rely on these underlying frameworks for asynchronous communication.[1]

3.1. pt2pt Component. The pt2pt component depends on the PML for all communication features and does not employ the optimizations available in the rdma component (discussed in Section 3.2). Originally, the pt2pt component was developed as a prototype to explore the implementation details of the MPI one-sided specification. The specification is particularly nuanced, and many issues in implementation do not become apparent until late in the development process. Most Open MPI users will never use the pt2pt component, as the rdma component is generally the default. However, the CM PML, developed shortly after the pt2pt component, does not use the BTL framework, meaning that the rdma OSC

[1]This design is problematic due to the lack of support for threaded communication within Open MPI; it is impossible for either component to be truly asynchronous without major advancement in the asynchronous support of the PML and BTL frameworks. The Open MPI community is expanding threaded progress support, but it will likely take many years to implement properly.

component is not available. Therefore, the pt2pt component is required in certain circumstances.

The pt2pt component translates every MPI_PUT, MPI_GET, and MPI_ACCUMULATE operation into a request sent to the target using an MPI_ISEND. Short put and accumulate payloads are sent in the same message as the request header. Long put and accumulate payloads are sent in two messages: the header and the payload. Because the origin process knows the size of the reply buffer, get operations always send the reply in one message, regardless of size.

Accumulate is implemented in two separate cases: the case where the operand is MPI_REPLACE, and all other operands. In the MPI_REPLACE case, the protocol is the same as for a standard put, but the window is locked from other accumulate operations during data delivery. For short messages, where the message body is immediately available and is delivered via local memory copy, this is not an issue. However, for long messages, the message body is delivered directly into the user buffer and the window's accumulate lock may be locked for an indeterminate amount of time. For other operations, the message is entirely delivered into an internal buffer. Open MPI's reduction framework is then used to reduce the incoming buffer into the existing buffer. The window's accumulate lock is held for the duration of the reduction, but does not need to be held during data delivery.

3.2. rdma Component. Three communication protocols are implemented for the rdma one-sided component: *send/recv*, *buffered*, and *RMA*. For networks which support RMA operations, all three protocols are available at run-time, and the selection of protocol is made per-message.

3.2.1. *send/recv.* The *send/recv* protocol performs all short message and request communication using the send/receive interface of the BTL, meaning data is copied at both the sender and receiver for short messages. Control messages for general active target synchronization and passive synchronization are also sent over the BTL send/receive interface. Long put and accumulate operations use use the PML. Communication is not started until the user ends the exposure epoch.

The use of the PML for long messages requires two transfers: one for the header over the BTL and one for the payload of the PML. The PML will likely then use a rendezvous protocol for communication, adding latency to the communication. This extra overhead was deemed acceptable, as the alternative involved duplicating the complex protocols of the OB1 PML (See [**80**]).

3.2.2. *buffered.* The *buffered* protocol is similar to the *send/recv* protocol. However, rather than starting a BTL message for every one-sided operation, messages are buffered during a given exposure epoch. Data is packed into an eager-sized BTL buffer, which is generally 1–4 KB in size. Messages are sent either when the buffer is full and the origin knows the target has entered an access epoch or at the end of the access epoch. Long messages are sent independently (no coalescing) using the PML protocol, although message headers are still coalesced.

The one-sided programming paradigm encourages short messages for communication, and most networks optimized for message passing are optimized for larger message transfers. Given this disparity, the *buffered* protocol provides an opportunity to send fewer larger messages than the *send/recv* protocol.

3.2.3. *RMA.* Unlike the *send/recv* and *buffered* protocols, the *RMA* protocol uses the RMA interface of the BTL for contiguous data transfers. All other data is transferred using the *buffered* protocol. MPI_ACCUMULATE also falls back to the *buffered* protocol, as NIC atomic support is premature and it is generally accepted that a receiver computes model offers the best performance. [**68**] Like the *buffered* protocol, communication is not started until confirmation is received that the target has entered an access epoch. During this time, the *buffered* protocol is utilized.

Due to the lack of remote completion notification for RMA operations, care must be taken to ensure that an epoch is not completed before all data transfers have been completed. Because ordering semantics of RMA operations tends to vary widely between network interfaces (especially compared to send/receive operations), the only ordering assumed by the rdma component is that a message sent after local completion of an RMA operation will

result in remote completion of the send after the full RMA message has arrived. Therefore, any completion messages sent during synchronization may only be sent after all RMA operations to a given peer have completed. This is a limitation in performance for some networks, but adds to the overall portability of the system.

4. Synchronization

Both the pt2pt and rdma utilize similar synchronization protocols. When a control message is sent, it is over the PML for pt2pt and the BTL's send/receive interface for the rdma component. The rdma component will buffer access epoch start control messages, but will not buffer access epoch completion control messages or exposure control messages.

4.1. Fence Synchronization. MPI_WIN_FENCE is implemented as a collective call to determine how many requests are incoming to complete the given epoch followed by communication to complete all incoming and outgoing requests. The collective operation is a MPI_REDUCE_SCATTER call, utilizing Open MPI's tuned MPI collective implementation. Each request is then started and two counters (number of outstanding incoming and outgoing requests) are maintained, with the call to MPI_WIN_FENCE not returning until both are 0.

If the assert MPI_MODE_NO_PRECEEDE is given to the fence call, it is a promise by the user that no communication occurred in any process during the last epoch. As it is already known that there are no incoming requests in this case, no requests need to be started and the collective operation is not performed. Verifying the number of scheduled operations is inexpensive, so the assertion is verified before the fence completes.

As mentioned previously, when using the *RMA* protocol in the rdma component, there is a completion issue that does not exist with other protocols. The target of an operation receives no notification that the RMA request has finished, so completing the exposure epoch is problematic. Ordering is maintained between RMA operations and send/receive operations in the BTL when specifically requested. A control message is sent to each peer when all RMA operations to that peer have started, with the number of RMA operations started to that peer. The count of operations is decremented by the number of RMA

operations. While the extra message is extra overhead, the other option when using RMA protocols is a complete barrier, which can be even more expensive.

4.2. General Active Target Synchronization. General active target synchronization, also known as Post/Wait/Start/Complete after the calls involved, allows the user to independently start access and exposure epochs on a subset of the window. MPI_WIN_START and MPI_WIN_COMPLETE start and complete an access epoch, while MPI_WIN_POST and either MPI_WIN_TEST or MPI_WIN_WAIT start and complete an exposure epoch.

MPI_WIN_START is not required to block until all remote exposure epochs have started, instead the implementation returns immediately and starts a local access epoch. Communication calls are buffered at least until a control message from the target has been received confirming the target is in an exposure epoch. During MPI_WIN_COMPLETE, all RMA operations are completed, then a control message with the number of incoming requests is sent to all peer processes. MPI_WIN_COMPLETE returns once the control message has completed (either by the PML or BTL).

MPI_WIN_POST sends a short control message to each process in the origin processes group notifying the process that the exposure epoch has started, then returns. MPI_WIN_WAIT blocks until it has received control messages from each process in the specified group and it has received all expected communication requests. It then completes the exposure epoch and returns. The call MPI_WIN_TEST is similar to MPI_WIN_WAIT, with the exception that it does not block and instead sets a status flag if the completion requirements have been met.

4.3. Passive Synchronization. Passive synchronization allows for true one-sided communication. The target process is not directly involved in communication or synchronization. In order to progress communication when the target process is not entering the MPI library, passive synchronization requires an asynchronous agent that can modify the target process's memory space. Both the pt2pt and rdma component relies on an the underlying transport for this asynchronous communication, as discussed in Section 2.

MPI_WIN_LOCK sends a control message requesting the target process start an exposure epoch and the call returns immediately, not waiting for an answer from the target. The target starts an exposure epoch upon reception of the control message if the lock can be obtained in the manner defined (either shared or exclusive). If the lock can not be obtained immediately, the request is queued until it can be satisfied. When the lock can be satisfied, a control message is sent back to the origin process.

MPI_WIN_UNLOCK waits for the control message from the target process that the exposure epoch on the target has started. It then sends a count of expected requests to the target process and starts all queued results. MPI_WIN_UNLOCK waits for local completion of communication requests to ensure that it is safe to allow those buffers to be reused, then returns. The target process will not release the lock and complete the exposure epoch until all requests are received and processed. At that time, the exposure epoch is ended and an attempt is made to satisfy any pending lock requests.

5. Performance Evaluation

The performance of both the pt2pt and rdma OSC components is demonstrated using latency and bandwidth micro-benchmarks, as well as a ghost-cell update kernel. Open MPI results are generated using the 1.3 release. MVAPICH2 0.9.8 results are also provided for comparison. Both implementations were compiled with the GNU Compiler Collection (GCC), version 4.1.2. No configuration or run-time performance options were specified for MVAPICH2 or Open MPI.

All tests were run on odin.cs.indiana.edu, a 128 node cluster of dual-core dual-socket 2.0 GHz Opteron machines, each with 4 GB of memory and running Red Hat Enterprise Linux 5. Nodes are connected with both 1 GB Ethernet and InfiniBand. Each node contains a single Mellanox InfiniHost PCI-X SDR HCA, connected to a 148 port InfiniBand switch. The InfiniBand drivers are from the Open Fabrics Enterprise Distribution, version 1.3.1.

5.1. Global Synchronization Performance. Figure 1 shows the cost of synchronization using the MPI_FENCE global synchronization mechanism. 98% of the cost of Open MPI's MPI_FENCE is spent in MPI_REDUCE_SCATTER. Unfortunately, MPI_REDUCE_SCATTER

currently has a higher cost in Open MPI than in MPICH2 and MVAPICH2. This is an area of active development in Open MPI and should be resolved in the near future. LAM/MPI's results are not shown because the cost of MPI_FENCE is ten times higher than the other MPI implementations.

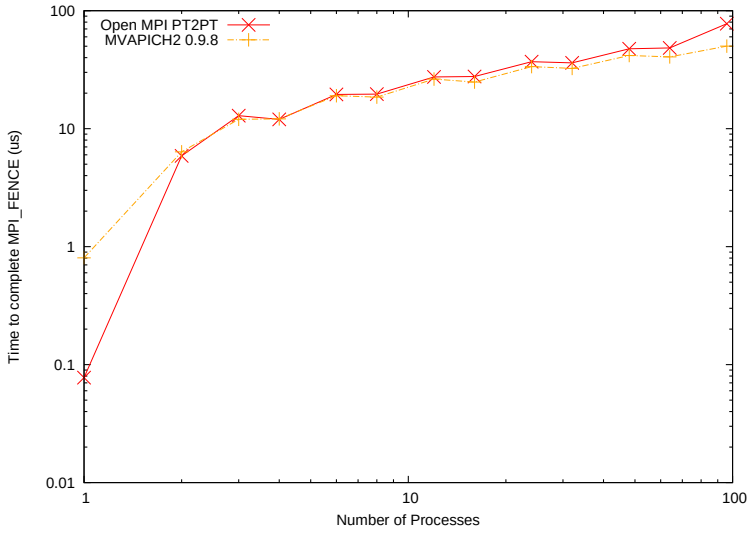
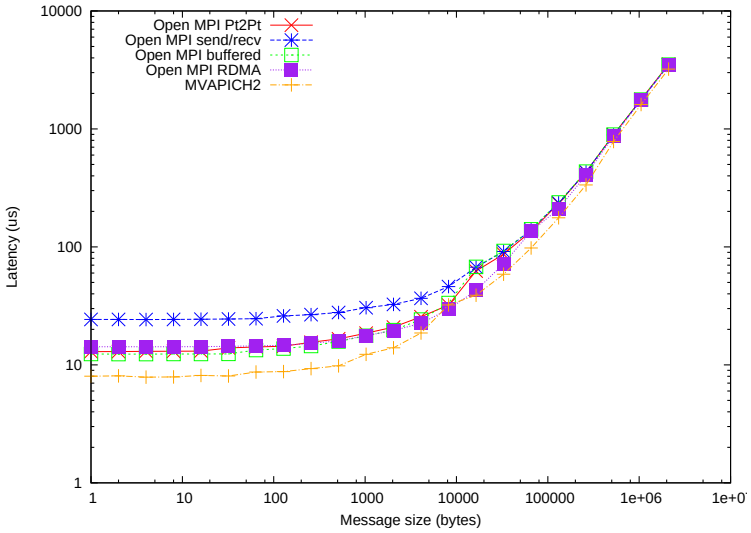


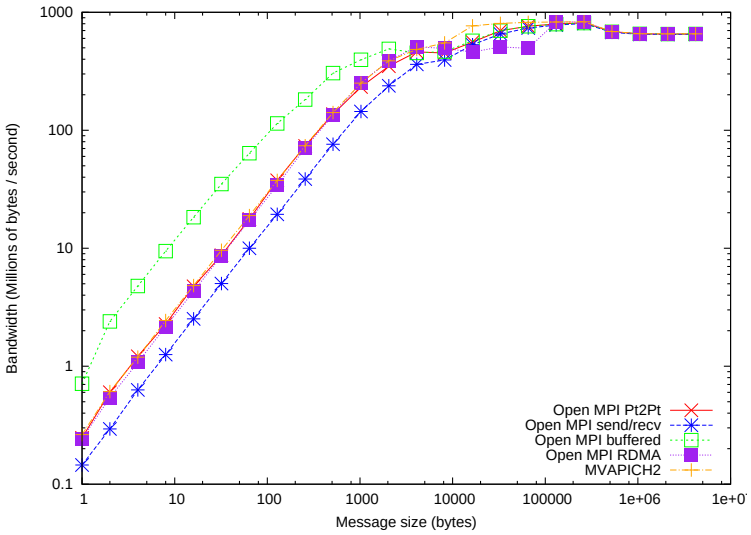
FIGURE 1. Cost of completing a Fence synchronization epoch, based on number of processes participating in the window.

5.2. Latency / Bandwidth Micro-benchmarks. The Ohio State benchmarks [**64**] were used to analyze both the latency and bandwidth of the the one-sided communication functions. The suite does not include a bandwidth test for MPI_ACCUMULATE, so those results are not presented here. All tests use generalized active synchronization.

Figure 2 presents the latency and bandwidth of MPI_PUT. The *buffered* protocol presents the best latency for Open MPI. Although the message coalescing of the *buffered* protocol does not improve performance of the latency test, due to only one message pending during an epoch, the protocol outperforms the *send/recv* protocol due to starting messages eagerly, as soon as all post messages are received. The *buffered* protocol provides lower latency than the *rdma* protocol for short messages because of the requirement for portable completion semantics, described in the previous section. No completion ordering is required for the *buffered* protocol, so MPI_WIN_COMPLETE does not wait for local completion of the data



(a) Latency



(b) Bandwidth

FIGURE 2. Latency and Bandwidth of MPI_PUT calls between two peers using generalized active synchronization.

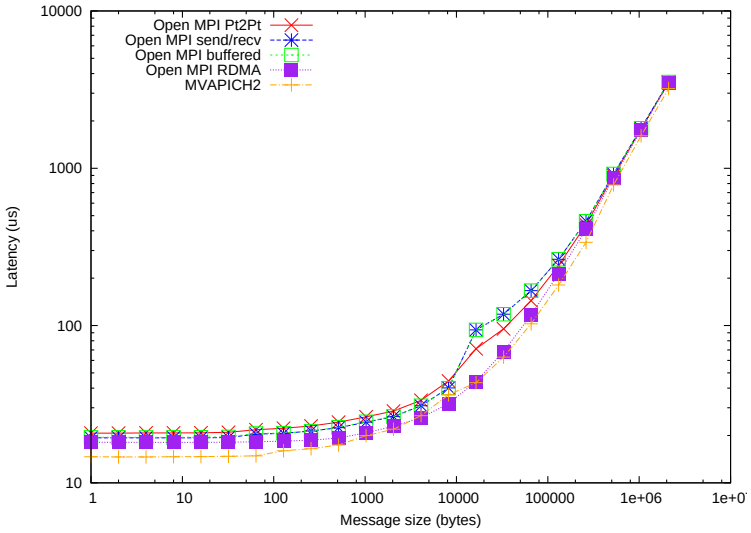
transfer before sending the completion count message. On the other hand, the *rdma* protocol must wait for local completion of the event before sending the completion count control message, otherwise the control message could overtake the RDMA transfer, resulting in erroneous results.

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

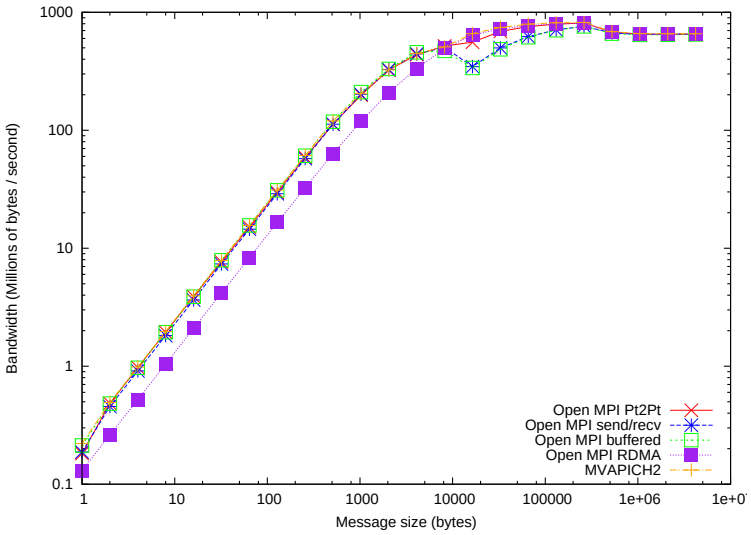
The latency and bandwidth of MPI_GET are shown in Figure 3. The *rdma* protocol has lower latency than the send/receive based protocols, as the target process does not have to process requests at the MPI layer. The present *buffered* protocol does not coalesce reply messages from the target to the origin, so there is little advantage to using the *buffered* protocol over the *send/recv* protocol. For the majority of the bandwidth curve, all implementations other than the *rdma* protocol provide the same bandwidth. The *rdma* protocol clearly suffers from a performance issue that the MVAPICH2 implementation does not. For short messages, we believe the performance lag is due to receiving the data directly into the user buffer, which requires registration cache look-ups, rather than copying through a pre-registered "bounce" buffer. The use of a bounce buffer for MPI_PUT but not MPI_GET is an artifact of the BTL interface, which will be addressed in the future.

MPI_ACCUMULATE, when the operation is not MPI_REPLACE, requires target side processing for most interconnects, including InfiniBand. For reasons similar to MPI_PUT, the latency of Open MPI's MPI_ACCUMULATE is slightly higher than that of MVAPICH2, as seen in Figure 4. Similar to MPI_PUT, however, the ability to handle a large number of messages is much greater in Open MPI's bundling implementation than is MVAPICH2, which is likely to be much more critical for real applications.

5.3. Ghost-Cell Exchange. The MPI community has not standardized on a set of "real world" benchmarks for the MPI one-sided interface, but the ghost-cell update was first used in [**87**] and later added to the `mpptest` suite from Argonne National Laboratory [**38**].



(a) Latency



(b) Bandwidth

FIGURE 3. Latency and Bandwidth of MPI_GET calls between two peers using generalized active synchronization.

An example of the ghost-cell exchange kernel using fence synchronization is shown in Figure 5. The implementation for general active target synchronization is similar, although setting up the groups for communication is more complex.

Figures 6 and 7 show the cost of performing an iteration of a ghost cell update sequence. The tests were run across 32 nodes, one process per node. For both fence and generalized

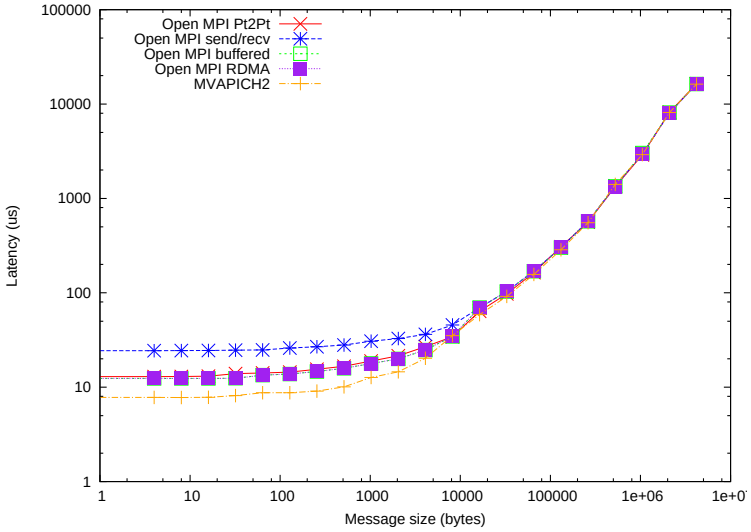


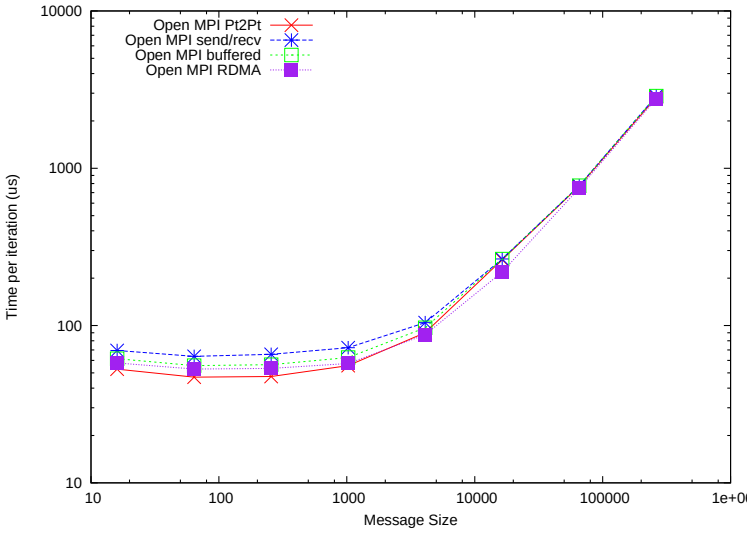
FIGURE 4. Latency of MPI_ACCUMULATE calls using MPI_SUM over MPI_INT datatypes and generalized active synchronization.

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

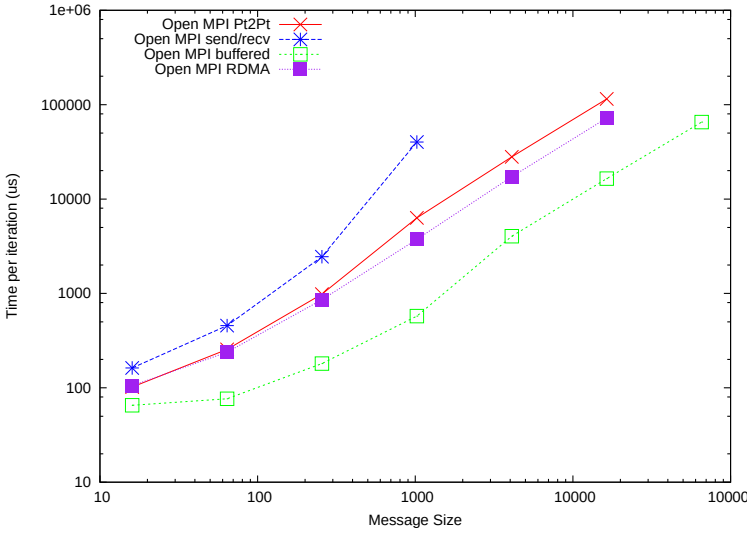
FIGURE 5. Ghost cell update using MPI_FENCE

active synchronization, the ghost cell update with large buffers shows relative performance similar to the put latency shown previously. This is not unexpected, as the benchmarks are similar with the exception that the ghost cell updates benchmark sends to a small number of peers rather than to just one peer. Fence results are not shown for MVAPICH2 because the tests ran significantly slower than expected and we suspect that the result is a side effect of the testing environment.

When multiple puts are initiated to each peer, the benchmark results show the disadvantage of the *send/recv* and *rdma* protocol compared to the *buffered* protocol. The number of messages injected into the MPI layer grows as the message buffer grows. With larger buffer sizes, the cost of creating requests, buffers, and the poor message injection rates of



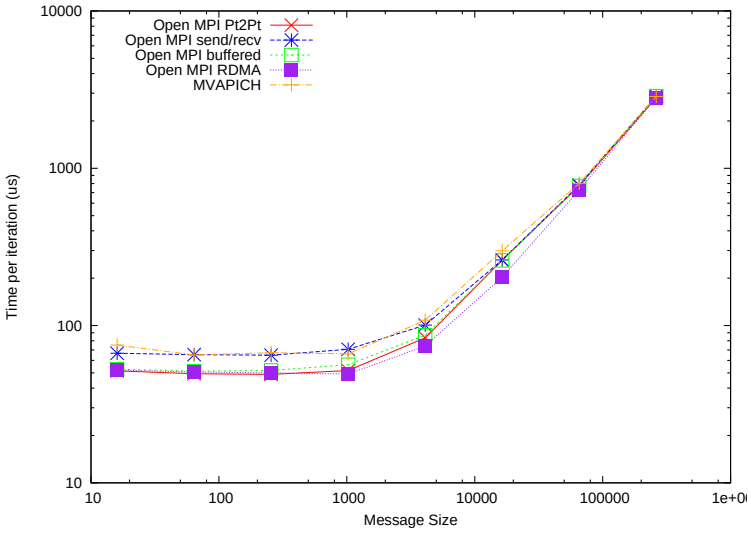
(a) one put



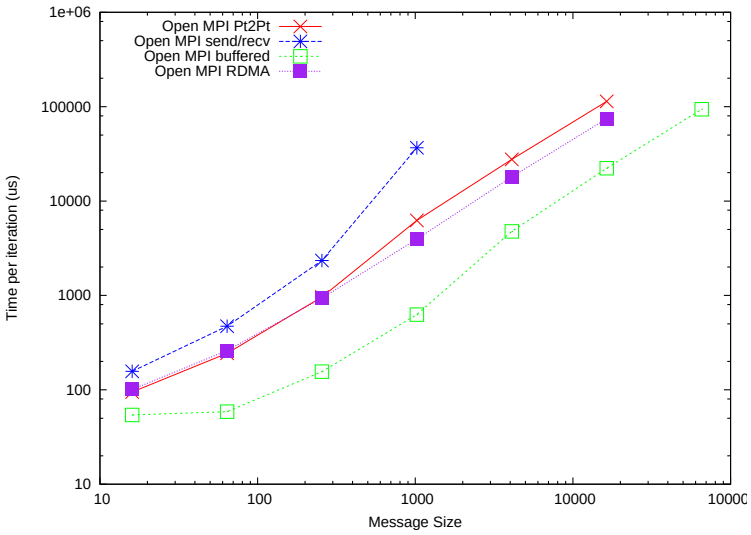
(b) many puts

FIGURE 6. Ghost cell iteration time at 32 nodes for varying buffer size, using fence synchronization.

InfiniBand becomes a limiting factor. When using InfiniBand, the *buffered* protocol is able to reduce the number of messages injected into the network by over two orders of magnitude.



(a) one put



(b) many puts

FIGURE 7. Ghost cell iteration time at 32 nodes for varying buffer size, using generalized active synchronization.

6. Conclusions

As we have shown, there are a number of implementation options for the MPI one-sided interface. While the general consensus in the MPI community has been to exploit the RMA interface provided by modern high performance networks, our results appear to indicate that

such a decision is not necessarily correct. The message coalescing opportunities available when using send/receive semantics provides much higher realized network bandwidth than when using RMA due to the higher message rate. The completion semantics imposed by a portable RMA abstraction also requires ordering that can cause higher latencies for RMA operations than for send/receive semantics.

Using RMA operations has one significant advantage over send/receive; the target side of the operation does not need to be involved in the message transfer, so the theoretical availability of computation/communication overlap is improved. In our tests, we were unable to see this in practice, likely due less to any shortcomings of RMA and more due to the two-sided nature of the MPI one-sided interface. Further, we expected the computation/communication overlap advantage to become less significant as Open MPI develops a stronger progress thread model, allowing message unpacking as messages arrive, regardless of when the application enters the MPI library.

CHAPTER 8

Conclusions

This thesis has demonstrated the applicability of the one-sided paradigm to a large class of applications. More importantly, it has demonstrated a practical taxonomy of both one-sided implementations and applications. A number of important conclusions for one-sided interfaces may be drawn from the previous chapters and are presented in Section 1. In addition, we believe that these lessons learned are applicable to other programming paradigms, and we discuss this further in Section 3. Final thoughts are then presented in Section 4.

1. One-Sided Improvements

The one-sided communication paradigm has a number of advantages for many application classes. In addition to high performance, the one-sided paradigm also offers more straight-forward implementation patterns for the application programmer. If we assume a correlation between code length and complexity within the same algorithm, this difference in implementation length suggests the one-sided implementation of the connected components and PageRank algorithms are simpler than the message passing implementations. While demonstrably useful, the one-sided paradigm does have a number of shortcomings, many of which must be addressed in future changes to interfaces and implementations.

Non-blocking operations are critical for high performance when using one-sided communication. Non-blocking provides an ideal method for covering the high relative latency of modern communication networks, provided there is enough work to allow multiple operations to be outstanding at any given time. The non-blocking interface does introduce complexity, particularly when programmers must extensively search for available parallelism. Both the connected components and PageRank algorithms provide a high degree

of parallelism via long adjacency lists for interesting graph types. For example, Figure 1 demonstrates the PageRank core from Chapter 5 using non-blocking get calls instead of the original blocking calls.

```
BGL_FORALL_VERTICES_T(v, g, Graph) {
  put(from_rank, v, get(from_rank, v) / out_degree(v, g));
}
shmem_barrier_all();
BGL_FORALL_VERTICES_T(v, g, Graph) {
  rank_type rank(0);
  double *rets = new double[indegree(v, g)];
  double *current_ret = rets;
  BGL_FORALL_INEDGES_T(v, e, g, Graph) {
    shmem_double_get(current_ret++,
                     from_rank.start() + local(source(e, g)),
                     1, get(owner, source(e, g)));
  }
  shmem_fence();
  for (int i = 0 ; i < indegree(v, g) ; ++i) {
    rank += rets[i];
  }
  delete [] rets;
  put(to_rank, v, (1 - damping) + damping * rank);
}
shmem_barrier_all();
```

FIGURE 1. Cray SHMEM implementation of the PageRank update step, using a bi-directional graph and the "pull" algorithm with a non-blocking get operation.

The data from Chapters 4 and 5 suggest that the set of atomic operations provided by an one-sided interface drives its applicability to many problems. In the case of connected components, MPI one-sided is unusable due to the lack of any calls which atomically return the value of the updated address. Due to the relatively higher latency and ability of NICs to perform calculations on the target node, there is not an equivalence between Atomic Operate and Atomic Fetch and Operate operations, as there is with local memory operations. Further, the datatypes supported for arithmetic operations is crucial to the general success of a particular one-sided interface. The PageRank implementations demonstrate the

importance of a wide set of atomic arithmetic operations, as Cray SHMEM's performance is partly limited by the inability to perform floating point atomic operations.

It has also been shown that there is also a performance implication in the choice of atomic operations which a one-sided interface provides. The body of work proving the universality of compare-and-swap, fetch-and-add, and load locked/store conditional still hold from a correctness standpoint. However, the performance of remote atomic operations involves such a high latency that the correct choice is essential. For example, an Atomic Fetch and Operate when correctly implemented involves a single round trip to the remote host (although the remote host may invoke multiple operations to local memory to complete the operation), but implementing Atomic Fetch and Operate using Compare and Swap may involve multiple round trip messages between nodes. The high latency of the network round trip dictates a much difference performance characteristic between the two designs.

These insights lead us to the conclusion that a general one-sided implementation should provide a rich set of operations if it is to successfully support the widest possible application set. These operations include both blocking and non-blocking communication calls. It also includes a richer set of atomic operations than can be found in any existing one-sided implementation. These include Atomic Operate, Atomic Fetch and Operate, Compare and Swap, and Atomic Swap, with the arithmetic operations defined for a variety of integer sizes, as well as single and double precision floating point numbers. Even with the small application set studied in this thesis, we have seen applications that require such a rich set of primitives.

While not apparent in the case studies presented, existing one-sided implementations are limited in the address regions which can be used as the target for communication. This is unlike current message passing libraries, which are generally able to send or receive from any valid memory address. MPI one-sided communication is limited to windows, which are created via a collective operation. Less flexible is Cray SHMEM, which requires communication be targeted into the symmetric heap. This limitation will be most pronounced when building libraries which utilize one-sided interfaces, which may not be able to impose such restrictive memory access patterns on an application.

Not discussed in the case studies is the benefit of utilizing registers instead of memory locations for the origin side data. The Cray T3D was able to efficiently support this model of communication utilizing the e-registers available on the platform, although the Cray SHMEM interface originally developed for that platform has since shed the ability to use registers for communication. The ARMCI interface provides API suppose for register-based communication, although it is unclear how much performance advantage such an API call currently provides. Modern interconnects are largely designed to use DMA transfers to move data (even headers) from host memory to network interface, so storing data to memory before communication is required. Network interfaces may return to using programmed-I/O style communication in order to improve message rates, however, leading to a return in the performance advantage to register-targeted communication. If such a situation occurs, it would be necessary to further extend a general one-sided interface to include sending from and receiving to registers instead of memory.

2. MPI-3 One-Sided Effort

The MPI Forum, which is responsible for the MPI standardization effort, has recently begun work on MPI 3.0. It is likely that MPI 3.0 will attempt to update the MPI one-sided communication specification. Currently, plans including a specification for an atomic fetch and operate operation, in addition to MPI_ACCUMULATE, as well as plans for fixing the heavy-weight synchronization infrastructure. There has also been discussion about how to eliminate the collective window creation for applications which need to access large parts of the virtual address space.

The addition of atomic operations other than MPI_ACCUMULATE would solve the problems with connected components described in Chapter 4. Although an atomic fetch and operate function would allow implementation of the connected components algorithm, a compare and swap operation would allow for a straight-forward implementation similar to the Cray SHMEM implementation. An atomic fetch and operate implementation requires a much more complex implementation in which the atomic operation is used to mark components as visited or not visited, and then further work is performed to handle the proper

marking of components. This suggests that adding an atomic fetch and operate function is insufficient and a compare and swap operation is also critical.

Although the active synchronization mechanisms are effective for applications with high global communication or well known communication patterns, it can be problematic for pure one-sided applications. At the same time, the passive synchronization mechanism incurs a high cost. This is because a round-trip is required, even for a single put operation. A connected components implementation in one-sided with a new compare and swap operation would also be impacted by the current passive target synchronization, as two round trip communication calls would be required (one for the lock, one for the compare and swap). A straight-forward solution would be a passive synchronization call which does not guarantee any serialization, but does open the required epochs. Epoch serialization is not required if atomic operations are used, as the atomic operations provide the required serialization.

Finally, the global creation of windows, while straight forward, causes problems for applications which must communicate with the entire remote address space. A recent proposal includes the creation of a pre-defined MPI_WIN_WORLD which encompasses the entire address space. One disadvantage of such a proposal is that the entire address space is always available for communication, which complicates the use of communication interfaces which limit the amount of memory which can be simultaneously used for communication. Another possibility would be to remove the collective creation requirement, which would push the problem of communication regions to the upper level, which is likely to have more knowledge about which memory is to be used for communication.

3. Cross-Paradigm Lessons

A number of the lessons learned in this thesis for the one-sided programming model can be applied to other programming models. Many paradigms outside of message passing are based on a similar set of communication primitives, including active messages, work queues, and one-sided. We first look at the implications of this thesis on the work queue and active message communication primitives. We then examine the UPC partitioned global address space language, CHARM++, and ParalleX programming paradigms.

3.1. Active Messages and Work Queues. Active Messages, initially described in Chapter 2, provides a sender-directed communication mechanism. The receiver does not explicitly receive a message, but invokes a function upon reception of a new message. The designers of Active Messages envisioned hardware support to allow fast interrupt handlers which could execute message handlers. Current hardware does not provide such a mechanism, and interrupts take hundreds of thousands of cycles to process, even when the kernel/user space boundaries are ignored.

Many recent incarnations of active message style programming, including GASNet, utilize a work queue model to replace the interrupt mechanism. In the work queue model, the sender inserts the message and context information on the target process's work queue. The receiver polls the work queue on a regular basis. Handler functions are then triggered from the polling loop without an interrupt or context switch. On modern hardware, the work queue offers much higher performance than interrupt driven handling. In addition to supporting active messages, the work queue model can also be used directly, as is the case with ParalleX.

Work queue primitives pose a number of challenges for modern network design not found in one-sided models. In particular, work queues require a receiver-directed message delivery or non-scalable memory usage. Therefore, it is unlikely adequate message rates can be achieved on scalable systems without significant specialized queue management hardware between the NIC and processor. Further complicating the work queue requirements is the need for advanced flow control. As the queue must be emptied by the application, it is possible for a queue to overflow during computation phases. Traditional flow control methods are either non-scalable (credit based) or bandwidth intensive (NACKs/retries). In a high message rate environment, current scalable flow control methods make the network highly susceptible to congestion, which pose additional network design challenges. Potential solutions include NIC hardware which interrupts the host processor when the work queue reaches some preset high water mark.

3.2. UPC. The most prevalent implementation of the UPC specification is the Berkeley UPC compiler. The run-time for Berkeley UPC utilizes the GASNet communication layer for data movement, and therefore inherits many of the problems faced by both active messages and one-sided implementations. However, because the compiler, not the user, is adding explicit communication calls, the overflow problem can be mitigated by polling the work queue more heavily in areas of the code during which overflow is likely. Communication hot-spots for high-level constructs such as reductions are still possible, although a sufficiently advanced compiler should be able to prevent such hot-spots through the use of transformations to logarithmic communication patterns.

GASNet presents a rich one-sided interface, which is capable of transfers into any valid virtual address on the target process. Combined with the requirement of an active messages interface for communication, such an ability presents problems for layering the one-sided API in GASNet over either MPI one-sided or Cray SHMEM. Both interfaces greatly restrict the virtual address ranges which are valid for target side communication. Such restrictions are not unique to MPI or SHMEM, as most low-level device interfaces restrict addresses which can be used for communication to those which have explicitly been *registered* before communication. Significant work was invested in development of an efficient registration framework within GASNet to reduce the impact of this requirement. [**10**] It is unclear how such results could be applied to either MPI one-sided (due to the collective registration call) or SHMEM (due to the symmetric heap limitation).

3.3. CHARM++. CHARM++ [**49**] is an object oriented parallel programming paradigm based on the C++ language. CHARM++ is based on the concept of *chares*, parallel threads of execution which are capable of communicating with other chares. Chares can communicate either via message passing or via special communication objects. CHARM++ applications must be compiled with the CHARM++ compiler/preprocessor and are linked against a run-time library which provides communication services. The run-time also provides CHARM++'s rich set of load balancing features. The AMPI [**43**] project from the

authors of CHARM++ provides a multi-threaded, load balancing MPI implementation on top of CHARM++.

Like Berkeley UPC, CHARM++ is capable of mitigating flow control issues inherent in the work queue model due to the compiler/preprocessor's ability to add queue draining during periods of potential communication. Further, the run-time library's rich load balancing features should help mitigate the computation hot-spot issues which are likely to occur in many unbalanced applications. The communication patterns AMPI is used, rather than using CHARM++ directly, should be similar to traditional message passing implementations, meaning that although it will have to perform message matching, it will also tend to send few, large messages.

3.4. ParalleX. ParalleX [**30**] is a new model for high performance computing which offers improved performance, machine efficiency, and easy of programming for a variety of application domains. ParalleX achieves these goals through a partitioned global address space combined, multi-threading, and a unique communication model. ParalleX extends the semantics of Active Messages with a *continuation* to define what happens after the action induced by the message occurs. Unlike traditional Active Messages, these *Parcels* allow threads of control to migrate throughout the system based both on data locality and resource availability. Although portions of the ParalleX model have been implemented in the LITL-X and DistPX projects, the model has not been fully implemented. The remainder of this section discusses issues intrinsic to the model and not to any one implementation.

ParalleX intrinsically solves a number of problems posed by the one-sided communication models described in this thesis. The global name space provided by ParalleX solves the memory addressing problems presented in Chapter 3, and which are likely to become more severe as data sets become more dynamic through an application lifespan. Light-weight multi-threading minimizes the effect of blocking communication calls, as new threads of context are able to cover communication latency. Finally, the migration of thread contexts to the physical domain of the target memory location simplifies many of the synchronization

and atomic operation requirements previously discussed. While a rich set of atomic primitives are still likely to be required for application performance, thread migration ensures that they occur in the same protection domain, moving hardware requirements from the network to processor.

The Parcels design, however, does raise a number of concerns. The case studies presented in Chapters 4 and 5 suggest a high message rate is required to satisfy the needs of informatics applications with the one-sided communication model. If we assume a similar number of Parcels will be required to migrate thread contexts for remote operations, a similarly high message rate will be required for ParalleX. Unlike latency, it is unlikely that the lightweight threading will be able to cover limitations in network message rates. Parcels utilize a work queue primitive for communication, and are susceptible to many of the work queue problems. The queue overflow and flow control contention issues likely mean that ParalleX is susceptible to data hot-spots, a problem which plagues the few custom multi-threaded informatics machines currently available.

4. Final Thoughts

One-sided communication interfaces rely heavily on underlying hardware for performance and features, perhaps more than any other communication paradigm. Many of the conclusions reached in Section 1 only increase the total feature set required of network hardware. Emulating one-sided communication with a thread running on the main processor limits performance due to caching effects and limits on the ability of NICs to wake up main processor threads. Therefore, it is reasonable to conclude that the future of the one-sided paradigm is tied future hardware designs and their ability to support a complex one-sided interface.

Future architectures will likely provide a number of communication paradigms, including message passing and one-sided interfaces. The choice of interface will hopefully be left to the programmer, based on the particular requirements of an application. While such choice is ideal, it does place the added burden of making communication paradigm choices on the application programmer. Therefore, accurate guidance on programming paradigm choices,

based on research rather than lore, is critical to the future of HPC systems. This thesis seeks to provide one piece of that puzzle, a detailed examination of the one-sided communication paradigm from the perspectives of both the communication library and the application. Literature already provides a fairly rich examination of the message passing paradigm, although work remains in determining when message passing is the correct choice for a given application. Similar examinations of developing and future programming paradigms are likewise necessary to drive future developments in HPC applications.

Bibliography

[1] Advanced Micro Devices. AMD Multi-Core White Paper, February 2005.

[2] Gail Alverson, Preston Briggs, Susan Coatney, Simon Kahan, and Richard Korry. Tera Hardware-Software Cooperation. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, New York, NY, USA, 1997. ACM.

[3] Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. Performance Analysis of MPI Collective Operations. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005) - Workshop 15*, 2005.

[4] Argonne National Laboratory. MPICH2. `http://www-unix.mcs.anl.gov/mpi/mpich2/`.

[5] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[6] Baruch Awerbuch and Yossi Shiloach. New Connectivity and MSF Algorithms for Ultracomputer and PRAM. In *Proceedings of the International Conference on Parallel Processing*, pages 175–179, 1983.

[7] Brian W. Barrett, Galen M. Shipman, and Andrew Lumsdaine. Analysis of Implementation Options for MPI-2 One-Sided. In *Proceedings, 14th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 242–250, Paris, France, October 2007. Springer-Verlag.

[8] Brian W. Barrett, Jeffrey M. Squyres, and Andrew Lumsdaine. Implementation of Open MPI on Red Storm. Technical Report LA-UR-05-8307, Los Alamos National Laboratory, Los Alamos, NM, October 2005.

[9] Brian W. Barrett, Jeffrey M. Squyres, Andrew Lumsdaine, Richard L. Graham, and George Bosilca. Analysis of the Component Architecture Overhead in Open MPI. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 175–182, Sorrento, Italy, September 2005. Springer-Verlag.

[10] Christian Bell and Dan Bonachea. A New DMA Registration Strategy for Pinning-Based High Performance Networks. In *17th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003)*, April 22-26 2003.

[11] Jonathan Berry, Bruce Hendrickson, Simon Kahan, and Petr Konecny. Software and Algorithms for Graph Queries on Multithreaded Architectures. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), Workshop on Multithreaded Architectures and Applications*, 2007.

[12] Johan Bollen, Marko A. Rodriguez, and Herbert Van de Sompel. Journal Status. *Scientometrics*, 69(3), 2006.

[13] Dan Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, University of California, Berkeley, October 2002.

[14] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. *International Journal of High Performance Computing and Networking*, 1(1-3):91–99, 2004.

[15] S. Booth and F. E. Mourao. Single Sided Implementations for SUN MPI. In *Proceeding of the 2000 ACM/IEEE Conference on Supercomputing*, 2000.

[16] Ron Brightwell, Tramm Hudson, Arthur B. Maccabe, and Rolf Riesen. The Portals 3.0 Message Passing Interface. Technical Report SAND99-2959, Sandia National Laboratory, November 1999.

[17] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the seventh international conference on World Wide Web 7*, pages 107–117, Amsterdam, The Netherlands, 1998. Elsevier Science Publishers B. V.

[18] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[19] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[20] David Callahan and Burton Smith. A future-based parallel language for a general-purpose highly-parallel computer. In *Selected papers of the second workshop on Languages and compilers for parallel computing*, pages 95–113, London, UK, UK, 1990. Pitman Publishing.

[21] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[22] Vinton Cerf, Yogen Dalal, and Carl Sunshine. RFC 675: Specification of Internet Transmission Control Program. IETF Network Working Group, December 1974.

[23] Lei Chai, Ping Lai, Hyun-Wook Jin, and Dhabaleswar K. Panda. Designing an Efficient Kernel-Level and User-Level Hybrid Approach for MPI Intra-Node Communication on Multi-Core Systems. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 222–229, Washington, DC, USA, 2008. IEEE Computer Society.

[24] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM Data Mining 2004*, Orlando, Florida, 2004.

[25] Cray, Inc. *Cray XMT System Overview*. 2008.

[26] Cray Research, Inc. *Cray T3E C and C++ Optimization Guide*. 1994.

[27] Paul Erdös and Alfred Rényi. On Random Graphs. *Publicationes Mathematicae Debrecen*, (6):290–297, 1959.

[28] Graham E. Fagg, Antonin Bukovski, and Jack J. Dongarra. HARNESS and Fault Tolerant MPI. *Parallel Computing*, 27:1479–1496, 2001.

[29] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 97–104, Budapest, Hungary, September 2004. Springer-Verlag.

[30] Guang R. Gao, Thomas Sterling, Rick Stevens, Mark Hereld, and Weirong Zhu. ParalleX: A Study of A New Parallel Computation Model. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, 2007.

[31] Al Geist, William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message-Passing Interface. In *Euro-Par '96 Parallel Processing*, pages 128–135. Springer Verlag, 1996.

[32] Patrick Geoffray. A Critique of RDMA. In *HPCWire*. August 18 2006. http://www.hpcwire.com/hpc/815242.html.

[33] Maciej Golebiewski and Jesper-Larsson Träff. MPI-2 One-sided Communications on a Giganet SMP cluster. In *Proceedings, 8th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science. Springer-Verlag, September 2001.

[34] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A Network-Failure-Tolerant Message-Passing System for Terascale Clusters. *International Journal of Parallel Programming*, 31(4), August 2003.

[35] Douglas Gregor and Andrew Lumsdaine. Lifting Sequential Graph Algorithms for Distributed-Memory Parallel Computation. In *Proceedings of the 2005 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, pages 423–437, October 2005.

[36] Douglas Gregor and Andrew Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.

[37] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI: The Complete Reference: Volume 2, the MPI-2 Extensions*. MIT Press, 1998.

[38] William Gropp and Rajeev Thakur. Revealing the Performance of MPI RMA Implementations. In *Proceedings, 14th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 272–280, Paris, France, October 2007. Springer-Verlag.

[39] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, 1991.

[40] Michael A Heroux. Design issues for numerical libraries on scalable multicore architectures. *Journal of Physics: Conference Series*, 125:012035 (11pp), 2008.

[41] P. N. Hilfinger, Dan Bonachea, Kaushik Datta, David Gay, Susan Graham, Amir Kamil, Ben Liblit, Geoff Pike, Jimmy Su, and Katherine Yelick. Titanium language reference manual. Technical Report UCB/EECS-2005-15.1, University of California, Berkeley, August 2006.

[42] W. Daniel Hillis. *The connection machine*. MIT Press, Cambridge, MA, USA, 1989.

[43] Chao Huang, Orion Lawlor, and Laxmikant V. Kale. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03*, pages 306–322, 2003.

[44] Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, Qi Gao, and Dhabaleswar K. Panda. Design and Implementation of High Performance MVAPICH2: MPI2 over InfiniBand. In *International Sympsoium on Cluster Computing and the Grid (CCGrid)*, Singapore, May 2006.

[45] InfiniBand Trade Association. InfiniBand Architecture Specification Vol 1. Release 1.2, 2004.

[46] Intel Corporation. The Intel iPSC/2 system: the concurrent supercomputer for production applications. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 843–846, New York, NY, USA, 1988. ACM.

[47] Intel Corporation. Intel Multi-Core Processors: Leading the Next Digital Revolution, 2006.

[48] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K. Panda, Darius Buntinas, Rajeev Thakur, and William Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 68–76, Budapest, Hungary, September 2004. Springer-Verlag.

[49] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based On C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 91–108. ACM Press, 1993.

[50] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Mueller, and Michael M. Resch. Towards efficient execution of parallel applications on the Grid: porting and optimization issues. *International Journal of Grid Computing*, 1(2):133–149, 2003.

[51] Thilo Kielmann, Henri E. Bal, Sergei Gorlatch, Kees Verstoep, and Rutger F. H. Hofman. Network performance-aware collective communication for clustered wide-area systems. *Parallel Computing*, 27(11):1431 – 1456, 2001.

[52] Charles H Koelbel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA, 1994.

[53] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Dong Chen, Mark E. Giampapa, Philip Heidelberger, Michael Blocksome, Ahmad Faraj, Jeff Parker, Joseph Ratterman, Brian Smith, and Charles Archer. The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 94–103, New York, NY, USA, 2008. ACM.

[54] Anthony Lamarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 130–140. ACM Press, 1994.

[55] Lawrence Berkeley National Laboratory and UC Berkeley. Berkeley UPC User's Guide, Version 2.8.0, 2008.

[56] Alan M. Mainwaring and David E. Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Technical Report UCB/CSD-96-918, EECS Department, University of California, Berkeley, October 1996.

[57] Amith R. Mamidala, Rahul Kumar, Debraj De, and Dhabaleswar K. Panda. MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 130–137, Washington, DC, USA, 2008. IEEE Computer Society.

[58] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the Intel 80-core network-on-a-chip terascale processor. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[59] Timothy G. Mattson and Michael Wrinn. Parallel programming: can we PLEASE get it right this time? In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 7–11, New York, NY, USA, 2008. ACM.

[60] Meiko, Inc. The Meiko computing surface: an example of a massively parallel system. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 852–859, New York, NY, USA, 1988. ACM.

[61] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[62] Myricom, Inc. Myrinet Express (MX): A High-Performance, Low-Level, Message-Passing Interface for Myrinet, 2006.

[63] nCUBE, Inc. The NCUBE family of high-performance parallel computer systems. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 847–851, New York, NY, USA, 1988. ACM.

[64] Network-Based Computing Laboratory, Ohio State University. Ohio State Benchmark Suite. http://mvapich.cse.ohio-state.edu/benchmarks/.

[65] Jarek Nieplocha and Bryan Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems. In *Proceedings, 3rd Workshop on Runtime Systems for Parallel Programming of the International Parallel Processing Symposium*, volume 1586 of *Lecture Notes in Computer Science*, San Juan, Puerto Rico, April 1999. Springer Verlag.

[66] Jarek Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:197–220, 1996.

[67] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease, and Edoardo Apra. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, 2006.

[68] Jarek Nieplocha, Vinod Tipparaju, and Edoardo Apra. An Evaluation of Two Implementation Strategies for Optimizing One-Sided Atomic Reduction. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, 2005.

[69] Cynthia A Phillips. Parallel Graph Contraction. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 148–157, June 1989.

[70] Jelena Pješivac-Grbović, Graham E. Fagg, Thara Angskun, George Bosilca, and Jack J. Dongarra. MPI Collective Algorithm Selection and Quadtree Encoding. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 40–48, Bonn, Germany, September 2006. Springer-Verlag.

[71] J. Postel. RFC 768: User Datagram Protocol. IETF Network Working Group, August 1980.

[72] Quadrics, Ltd. Quadrics QsNet.

[73] Quadrics, Ltd. Shmem Programming Manual, 2004.

[74] Umakishor Ramachandran, Gautam Shah, S. Ravikumar, and Jeyakumar Muthukumarasamy. Scalability Study of the KSR-1. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 237–240, Washington, DC, USA, 1993. IEEE Computer Society.

[75] John H. Reif. Optimal Parallel Algorithms for Integer Sorting and Graph Connectivity. Technical Report TR-08-85, Harvard University, March 1985.

[76] Francesco Romani. Fast PageRank Computation Via a Sparse Linear System (Extended Abstract).

[77] Sandia National Laboratories. Mantevo Project. `https://software.sandia.gov/mantevo/index.html`.

[78] Gopalakrishnan Santhanaraman, Sundeep Narravula, Amith R. Mamidala, and Dhabaleswar K. Panda. MPI-2 One-Sided Usage and Implementation for Read Modify Write Operations: A Cast Study with HPCC. In *Proceedings, 14th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 251–259, Paris, France, October 2007. Springer-Verlag.

[79] Yossi Shiloach and Uzi Vishkin. An O(n^2log n) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[80] Galen M. Shipman, Tim S. Woodall, George Bosilca, Richard L. Graham, and Arthur B. Maccabe. High Performance RDMA Protocols in HPC. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 76–85, Bonn, Germany, September 2006. Springer-Verlag.

[81] Galen M. Shipman, Timothy S. Woodall, Richard L. Graham, Arthur B. Maccabe, and Patrick G Bridges. InfiniBand Scalability in Open MPI. In *Proceedings of the 20th IEEE International Parallel And Distributed Processing Symposium (IPDPS 2006)*, 2006.

[82] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[83] David B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and Answers About BSP. *Scientific Programming*, 6(3):249–274, Fall 1997.

[84] Marc Snir, Steve W. Otto, Steve Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[85] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September 2003. Springer-Verlag.

[86] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[87] Rajeev Thakur, William Gropp, and Brian Toonen. Optimizing the Synchronization Operations in Message Passing Interface One-Sided Communication. *International Journal of High Performance Computing Applications*, 19(2):119–128, 2005.

[88] Jesper-Larsson Träff, Hubert Ritzdorf, and Rolf Hempel. The Implementation of MPI-2 One-Sided Communication for the NEC SX-5. In *Supercomputing 2000*. IEEE/ACM, 2000.

[89] Keith Underwood and Ron Brightwell. The Impact of MPI Queue Usage on Message Latency. In *2004 International Conference on Parallel Processing*, Montreal, Canada, 2004.

[90] John D. Valois. Lock-free linked lists using compare-and-swap. In *In Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, 1995.

[91] Deborah A. Wallach, Dawson R. Engler, and M. Frans Kaashoek. ASHs: Application-Specific Handlers for High-Performance Messaging. In *SIGCOMM '96*, August 1997.

[92] Timothy S. Woodall, Richard L. Graham, Ralph H. Castain, David J. Daniel, Mitchel W. Sukalski, Graham E. Fagg, Edgar Gabriel, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian W. Barrett, and Andrew Lumsdaine. Open MPI's TEG point-to-point communications methodology: Comparison to existing implementations. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 105–111, Budapest, Hungary, September 2004.

[93] Joachim Worringen, Andreas Gäer, and Frank Reker. Exploiting transparent remote memory access for non-contiguous and one-sided-communication. In *Proceedings of 16th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2002), Workshop for Communication Architecture in Clusters (CAC 02)*, 2002.

[94] Jian Zhang, Phillip Porras, and Johannes Ullrich. Highly Predictive Blacklisting. In *Proceedings of USENIX Security '08*, pages 107–122, 2008.

Brian W. Barrett

CONTACT INFORMATION

Scalable System Software Group
Sandia National Laboratories
P.O. Box 5800 MS 1319
Albuquerque, NM 87185-1319

Phone: 505-284-2333
bwbarre@sandia.gov

RESEARCH INTERESTS

The design and implementation of high performance computing communication systems, including both advanced network interface designs and software communication paradigms. Research into advanced systems capable of supporting both traditional message passing applications and emerging graph-based informatics applications.

EDUCATION

Indiana University — Bloomington, IN
Ph.D. Computer Science — *March 2009*
Thesis: *One-Sided Communication for High Performance Computing Applications*
Advisor: Andrew Lumsdaine
Committee: Randall Bramley, Beth Plale, and Amr Sabry

Indiana University — Bloomington, IN
M.S. Computer Science — *August 2003*
Advisor: Andrew Lumsdaine

University of Notre Dame — Notre Dame, IN
B.S. Computer Science, cum Laude — *May 2001*

EXPERIENCE

Limited Term Employee — **Sandia National Laboratories**
October 2007 - present — *Albuquerque, New Mexico*
Research into advanced network design, particularly network interface adapters, for message passing. Research and development of large scale graph algorithms as part of the MTGL and PBGL graph libraries. Design of advanced computer architectures capable of supporting large scale graph informatics applications.

Technical Staff Member — **Los Alamos National Laboratory**
October 2006 - October 2007 — *Los Alamos, New Mexico*
Research and development work on the Open MPI implementation of the MPI standard. Focus on enhancements for the Road Runner hybrid architecture system, including high-performance heterogeneous communication.

Student Intern — **Los Alamos National Laboratory**
Summer 2006 — *Los Alamos, New Mexico*
Research and development work on the Open MPI implementation of the MPI standard. Implemented the MPI-2 one-sided specification within Open MPI. Co-developed a high performance point-to-point engine for interconnects that support MPI matching in the network stack. Implemented support for the Portals communication library within the new matching point-to-point engine.

Research Assistant — **Open Systems Laboratory**
Fall 2001 – Spring 2003, — *Indiana University, Bloomington*
Fall 2004 – present
Research work in high performance computing, particularly implementations of the Message Passing Interface (both LAM/MPI and Open MPI). Worked with the Parallel Boost Graph Library development team on extensions to the MPI one-sided interface to improve performance and scalability of the library's graph algorithms. Designed

and implemented support for the Red Storm / Cray XT platform, including support for the Catamount light-weight operating system and Portals communication library.

Programmer Analyst **Information Sciences Institute**
2003-2004 *University of Southern California*
Member of the Joint Experimentation on Scalable Parallel Processors team, extending large scale military simulation software to more efficiently utilize modern HPC clusters. Co-developed a new software routing infrastructure for the project, increasing scalability and failure resistance.

Student Intern **Sandia National Laboratories**
Summer 2002 *Albuquerque, New Mexico*
Worked with the Scalable Computing Systems organization on the parallel run-time environment for the Cplant clustering system. Developed a run-time performance metrics system for the Cplant MPI implementation.

Student Intern **Sandia National Laboratories**
Summer 2001 *Albuquerque, New Mexico*
Provided MPI support for the Alegra code development team. Investigated fault tolerance options for large scale MPI applications within the context of LAM/MPI.

Publications

Brian W. Barrett, Jonathan W. Berry, Richard C. Murphy, and Kyle B. Wheeler. Implementing a Portable Multi-threaded Graph Library: the MTGL on Qthreads. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009), Workshop on Multithreaded Architectures and Applications*, 2009.

Brian W. Barrett, Galen M. Shipman, and Andrew Lumsdaine. Analysis of Implementation Options for MPI-2 One-sided. In *Proceedings, 14th European PVM/MPI Users' Group Meeting*, Paris, France, September 2007.

Richard L. Graham, Ron Brightwell, Brian W. Barrett, George Bosilca, and Jelena Pješivac-Grbović. An Evaluation of Open MPI's Matching Transport Layer on the Cray XT. In *Proceedings, 14th European PVM/MPI Users' Group Meeting*, Paris, France, September 2007.

Galen M. Shipman, Ron Brightwell, Brian W. Barrett, Jeffrey M. Squyres, and Gil Bloch. Investigations on InfiniBand: Efficient Network Buffer Utilization at Scale. In *Proceedings, 14th European PVM/MPI Users' Group Meeting*, Paris, France, September 2007.

Richard L. Graham, Brian W. Barrett, Galen M. Shipman, Timothy S. Woodall and George Bosilca. Open MPI: A High Performance, Flexible Implementation of MPI Point-to-Point Communications. In *Parallel Processing Letters*, Vol. 17, No. 1, March 2007.

Ralph Castain, Tim Woodall, David Daniel, Jeff Squyres, and Brian W. Barrett. The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing. In *Future Generation Computer Systems*. Accepted for publication.

Christopher Gottbrath, Brian Barrett, Bill Gropp, Ewing Rusty Lusk, and Jeff Squyres. An Interface to Support the Identification of Dynamic MPI 2 Processes for Scalable Parallel Debugging. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Bonn, Germany, September 2006.

Richard L. Graham, Brian W. Barrett, Galen M. Shipman, and Timothy S. Woodall. Open MPI: A High Performance, Flexible Implementation of MPI Point-To-Point Communications. In *Proceedings, Clusters and Computational Grids for cientific Computing*, Flat Rock, North Carolina, September 2006.

Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, and George Bosilca. Open MPI: A High Performance, Heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

Brian W. Barrett, Ron Brightwell, Jeffrey M. Squyres, and Andrew Lumsdaine. Implementation of Open MPI on the XT3. *Cray Users Group* 2006, Lagano, Switzerland, May 2006.

Brian W. Barrett, Jeffrey M. Squyres, and Andrew Lumsdaine. Implementation of Open MPI on Red Storm. Technical report LA-UR-05-8307, Los Alamos National Laboratory, Los Alamos, New Mexico, USA, October 2005.

B. Barrett, J. M. Squyres, A. Lumsdaine, R. L. Graham, and G. Bosilca. Analysis of the Component Architecture Overhead in Open MPI. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg. The Open Run-Time Environment (OpenRTE): A Transparent Multi-Cluster Environment for High-Performance Computing. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.

Brian Barrett and Thomas Gottschalk. Advanced Message Routing for Scalable Distributed Simulations. In *Proceedings, Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*, Orlando, FL 2004.

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalski, G.E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. Open MPI's TEG Point-to-Point Communications Methodology: Comparison to Existing Implementations. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

Brian W. Barrett. Return of the MPI Datatypes. ClusterWorld Magazine, MPI Mechanic Column, 2(6):34–36, June 2004.

Brian Barrett, Jeff Squyres, and Andrew Lumsdaine. Integration of the LAM/MPI environment and the PBS scheduling system. In *Proceedings, 17th Annual International Symposium on High Performance Computing Systems and Applications*, Quebec, Canada, May 2003.

John Mugler, Thomas Naughton, Stephen L. Scott, Brian Barrett, Andrew Lumsdaine, Jeffrey M. Squyres, Benoit des Ligneris, Francis Giraldeau, and Chokchai Leangsuksun.

OSCAR Clusters. In *Proceedings of the Ottawa Linux Symposium (OLS'03)*, Ottawa, Canada, July 23-26, 2003.

Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In *LACSI Symposium*, October 2003.

Thomas Naughton, Stephen L. Scott, Brian Barrett, Jeffrey M. Squyres, Andrew Lumsdaine, Yung-Chin Gang, and Victor Mashayekhi. Looking inside the OSCAR cluster toolkit. Technical report in PowerSolutions Magazine, chapter HPC Cluster Environment, Dell Computer Corporation, November 2002.

Software

LAM/MPI (`http://www.lam-mpi.org/`) Open source implementation of the MPI standard.

Open MPI (`http://www.open-mpi.org/`) High performance open source implementation of the MPI standard, developed in collaboration by the developers of LAM/MPI, LA-MPI, and FT-MPI.

Mesh-based routing infrastructure for the RTI-s implementation of the HLA discrete event simulation communication infrastructure, providing plug-in replacement to the existing tree-based routing infrastructure.

Honors and Awards

Department of Energy High Performance Computer Science fellowship, 2001–2003.

Service

Secretary, Computer Science Graduate Student Association, Indiana University, 2002–2003

President, Notre Dame Linux Users Group, University of Notre Dame, 2000–2001