SHARED MEMORY OPTIMIZATIONS FOR DISTRIBUTED MEMORY PROGRAMMING MODELS

Andrew Friedley

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
September 2013

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

Andrew Lumsdaine, Ph.D.

Greg Bronevetsky, Ph.D.

Arun Chauhan, Ph.D.

Jeremy Siek, Ph.D.

Martin Swany, Ph.D.

September 13, 2013

Acknowledgements

I'd like to thank my committee: Andrew Lumsdaine, Greg Bronevetsky, Arun Chauhan, Jeremy Siek, and Martin Swany. Without them this work could not have been completed, and their input has made the work better as a whole.

Andrew Lumsdaine is most especially deserving of thanks, for providing more than adequate support throughout my entire graduate career. He provided me with an excellent work environment and total freedom to pursue my interests and ideas. I can't imagine having a better advisor or supervisor to work for.

Greg Bronevetsky is also most deserving of thanks, as he volunteered his time and energy to take me on and act as a mentor for the past several years. Greg made it possible to perform my work at LLNL, which has led to much collaboration and professional connection that would not have been possible otherwise.

Although not a member of the Ph.D. committee, Torsten Hoefler also had a crucial role in this work. I am thankful for his initial work on the HMPI project, and continuing support on all aspects of the research forming this dissertation.

I am thankful for the mentorship and friendship of Matthew Leininger, who was the first to introduce me to the national laboratories.

Thanks to Rebecca Schmitt and Kelsey Allen for all their support with administrative tasks. Their help made it possible for me to remain focused on my work while meeting all the necessary degree, travel, and employment requirements.

Finally I'd like to thank Mary Judith Cerezo, who patiently supported me through all the stress and time I invested to bring this work to completion.

SHARED MEMORY OPTIMIZATIONS FOR DISTRIBUTED MEMORY PROGRAMMING MODELS

Andrew Friedley

In the world of parallel programming, there are two major classes of programming models: shared memory and distributed memory. Shared memory models share all memory by default, and are most effective on multi-processor systems. Distributed memory models separate memory into distinct regions for each execution context and are most effective on a network of processors. Modern and future High Performance Computing (HPC) systems will contain multi- and many-core processors connected by a network, resulting in a hybrid shared and distributed memory environment. Neither programming model is ideal in both areas. Now and in the future, optimizing parallel performance for both memory models simultaneously is a major challenge.

MPI (Message Passing Interface) is the de-facto standard for distributed memory programming, but results in less than ideal performance when used in a shared memory environment. Message passing incurs overhead in the form of unnecessary data copying as well specific queuing, ordering, and matching rules. In this thesis, we will present a series of techniques that optimize MPI performance in a shared memory environment, thus helping to solve the challenge of optimizing parallel performance for both distributed and shared memory. We introduce the concept of a shared memory heap, in which dynamically allocated memory is shared by default on all MPI processes within a node. We then use that to transparently optimize message passing with two new data transfer protocols. Next, we propose an MPI extension for ownership passing, which eliminates data copying

overheads completely. Instead of copying data, we transfer control (ownership) of communication buffers. Finally, we explore how shared memory techniques can be applied in the context of MPI and the shared memory heap. Loop fusion is a new technique for combining the packing and unpacking code on two different MPI ranks to eliminate explicit communication. All of these techniques are implemented in a freely available software library named Hybrid MPI (HMPI). We experimentally evaluate our work using a variety of micro-benchmarks and mini-applications. In the mini-applications, communication performance is improved up to 46% by our data transfer protocols, 54% by ownership passing, and 63% by loop fusion.

Contents

List of Figures

List of Acronyms

API: Application Programming Interface
DMA: Direct Memory Access
DSM: Distributed Shared Memory
FFT: Fast Fourier Transform
FIFO: First In, First Out
FLOP: Floating Point OPeration
GPU: Graphics Processing Unit
HMPI: Hybrid Message Passing Interface
HPC: High Performance Computing
I/O AT: Input/Output Acceleration Technology
IP: Internet Protocol
LiMIC: Linux kernel module for MPI Intra-node Communication
LLNL: Lawrence Livermore National Laboratory
LULESH: Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
KNEM: Kernel Nemesis
MD: Molecular Dynamics
MCS: Mellor-Crummey and Scott
MPI: Message Passing Interface
MRU: Most Recently Used
NUMA: Non-Uniform Memory Architecture
OPI: Ownership Passing Interface
OS: Operating System
PACX-MPI: PArallel Computer eXtension MPI
PGAS: Partitioned Global Address Space
PSM: Performance Scaled Messages
POSIX: Portable Operating System Interface
SMP: Symmetric MultiProcessor
SMT: Simultaneous MultiThreading
SPMD: Single Program Multiple Data
TCP: Transmission Control Protocol

1

Introduction

Two major classes of programming models exist in the world of parallel computing. Shared memory models such as POSIX threads [**10**], OpenMP [**71**], and OpenCL [**67**] share all memory by default. Communication among execution contexts is implicit; primitives for synchronization include mutexes, semaphores, and atomic counters. On the other hand, distributed memory models assign separate address spaces to each execution context, sharing no memory by default. Communication is expressed explicitly, for example in terms of sends, receives, and collective patterns like broadcast and all-to-all when using the Message Passing Interface (MPI) [**66**].

Both programming models are most effective when matched with the corresponding shared and distributed memory hardware that they were designed for. Shared memory

is most efficient on single multi-processor systems, while a distributed memory model is most efficient on many single-processor systems connected by a network. Multi-core processors are now ubiquitous, resulting in systems consisting of many multi-processor systems connected by a network. Neither programming model is efficient for all parts of the system, leading to sub-optimal performance when a single model is used for the entire heterogeneous system.

One solution to programming for heterogenous systems is to combine the two programming models. A shared memory model is used within compute nodes and a distributed programming model is used across the network (e.g., MPI combined with OpenMP [**80**, **81**]), sometimes referred to as 'MPI+X'. Software written in this manner makes efficient use of both models of parallelism presented by the hardware, but combining multiple programming models results in overly complex code. In turn, complex code leads to more frequent bugs, errors, and increased maintenance effort.

Alternatively, a single programming model could be applied across the entire system, regardless of underlying hardware type. Although either model works well with its corresponding hardware design, neither is optimal for the contrasting class of hardware. Shared memory models (such as distributed shared memory [**2**]) lack an expression of locality to differentiate between 'local' and 'remote' memory necessary for optimal performance over a distributed memory network. Distributed memory models, with their explicit locality, are too restrictive on shared memory hardware, incurring communication overhead and unnecessary data duplication.

The Partitioned Global Address Space (PGAS) family of programming languages [**1**, **3**, **12**, **14**, **32**, **70**, **87**] take the shared model and introduce partitioning as a method for expressing locality. All memory is still accessible by all execution contexts, but there is a distinction between local and remote data locations. PGAS works well on both types of hardware as well as modern multi-core clusters, but has not seen the same widespread adoption of MPI.

In this dissertation, we begin with MPI, which is the de-facto instance of the distributed programming model, and extend it to perform efficiently on shared memory hardware.

Thus, we take a similar approach to PGAS, but start from the distributed memory model. Many codes are already written using MPI, representing a huge investment of time and effort that would be difficult to transfer to shared memory models like PGAS. Due to this existing body of work, improving the distributed memory programming model (MPI) will have a siginificant impact on parallel computing. First, our work introduces new intra-node communication protocols that provide transparent speedups to existing MPI applications. Second, we introduce an ownership passing extension that introduces shared memory to MPI to further reduce communication overhead.

All of our work utilizes a shared heap. Normally, memory allocations returned by `malloc()` et al. is private to that process. A shared memory heap instead returns memory that accessible by all participating processes within a node (shared memory domain). Memory is accessible at the same address on all processes, avoiding the need for any address translation. Some systems, like IBM's Blue Gene/Q, already provide this functionality **[62]**. Linux kernel extensions like XPMEM **[72, 88]** can provide a shared heap, but are not installed by default on most systems. As part of this dissertation, we developed an approach using a large `mmap()` region to implement a shared heap entirely in user space, allowing our optimizations and extensions to work on most UNIX and UNIX-like systems that otherwise lack shared heap functionality.

Our approach to implementing our optimizations and extensions is novel as well. Rather than modifying an existing MPI library, we designed a layered library that is linked in between the application and existing (unmodified) MPI library. We refer to our implementation as Hybrid MPI (HMPI). HMPI intercepts communication within a node and optimizes it, while passing all other functionality (e.g., inter-node communication) down to the underlying MPI. This approach allows us to implement and experiment with our ideas quickly and portably across multiple platforms and MPI implementations. In particular, we have experimented on commodity x86-based clusters with entirely open source software as well as Blue Gene/Q with a proprietary MPI implementation.

Currently, a pipelined two-copy protocol is used to implement message passing via shared memory. LiMIC [**60**, **89**] and KNEM [**28**, **59**] are Linux kernel extensions that provide kernel-driven single-copy across distributed address spaces, but the overhead of system calls make them unsuitable for small messages. A shared heap makes single-copy message passing trivial by using `memcpy()`, works well for all message sizes, and does not necessarily require system modifications. We have developed two novel communication protocols that further improve performance over single-copy for small and large messages.

We went beyond the MPI specification to consider how it can be extended to further take advantage of shared memory. Used as a replacement for message passing, our ownership passing technique eliminates copying during communication altogether by passing pointers to buffers rather than making copies. Ownership passing directly exposes memory shared by ranks within a node, enabling applications to bypass MPI 's remaining inherent performance overhead in shared memory.

To this end, we further explore the use of shared memory communication techniques in the context of HMPI. We show how existing shared memory synchronization primitives, used via the shared heap, can replace explicit MPI communication completely, avoiding the overhead of message queueing and matching. We introduce a new technique we refer to as loop fusion. When communicating sparse data, an application may serialize into a contiguous communication buffer, pass the buffer (message) via MPI, then deserialize. This process is also known as the "pack-communicate-unpack" programming pattern. Since we can now share memory between MPI processes within the same node, we can transform this design pattern to eliminate the communication phase altogether and fuse the pack and unpack phases into a single fused loop that moves sparse data directly from one rank to another.

Together, these optimizations demonstrate how to effectively utilize shared memory in a distributed memory programming model, specifically MPI.

1.1. Hybrid MPI

We have implemented a 'Hybrid' MPI (HMPI) library to investigate message passing techniques using a shared memory heap. Rather than modifying an existing MPI library or implementing a new one, we have taken the unique approach of layering HMPI on top of any existing MPI library. In this way, we are able to extend both open- and closed-source MPI implementations on multiple platforms without modification, achieving maximum portability. We implement only the portions of MPI we are interested in for research purposes (shared memory point-to-point communication), and defer to the underlying MPI for all other functionality (e.g., inter-node communication and and miscellaneous utility routines). In this dissertation, we focus on point-to-point communication techniques. S. Li et al. have been researching collective communication techniques **[56]** in HMPI .

1.1.1. Shared Memory Heap. Standard heap allocators that implement `malloc()` et al. provide memory that is only accessible to the calling process. In other words, the default is a 'private' memory allocator. A shared memory heap allocator, on the other hand, provides memory allocations that are accessible by multiple processes within the node (shared memory domain). Applications using a shared heap have the ability to read and write any memory allocated by any participating process. Synchronization primitives such as pthreads mutexes, semaphores, and assembly-level lock code work in the shared heap just as they would in a multi-threaded environment.

However, only heap memory is shared. All other memory regions are private to each process, most importantly static/global variables and the stack. The result is an environment where multi-threaded programming techniques are possible, but with greater safety from errors. Existing libraries that are not thread-safe (i.e., depend on state in static or global variables) work as-is. The (minor) drawback of private static/global variables is just that—to share data stored in these private regions, it must be copied into a shared heap allocation.

The shared heap is a simple concept that can be used for all sorts of parallel programming. Although the idea is not new, to our knowledge, we are the first to implement the

shared heap entirely in user space as well as exploit it in the context of MPI. In addition to traditional multi-threaded programming, it also has uses in PGAS language and MPI implementations. This dissertation focuses on the applications of a shared heap in the context of intra-node message passing with MPI . We are addressing the limitations of a distributed memory model (MPI) by carefully introducing shared memory concepts.

Several machines have shared heap functionality built in. On IBM's Blue Gene/Q systems, a shared heap can be enabled using the `BG_MAPCOMMONHEAP` environment variable [**62**]. SGI and Cray systems have the XPMEM [**5**, **72**, **88**] kernel extension that allows mapping of memory from one process into another.

For UNIX-based systems that do not provide such functionality by default, such as commodity Linux clusters, we have developed a user-space shared heap allocator. We take an existing memory allocator and configure it to only use a modified `sbrk()` routine to acquire memory. Instead of requesting private memory from the system, our `sbrk()` returns memory from a large shared memory region mapped to the same location on all participating processes using `mmap()`. The result is a small memory allocator library that overrides the system's default allocator (using weak symbols) to provide a shared heap. No administrative access or operating system extensions are required.

The following sections discuss multiple ways in which a shared heap can be utilized in the context of MPI (and implemented in HMPI) for improved performance when communicating in a shared memory environment.

1.2. Shared Memory Message Passing

Utilizing a shared memory heap, we have developed several new intra-node communication protocols for MPI . These new techniques are implemented transparently in HMPI , providing speedups to existing applications without modifying them. First, we can transfer data from the sender's buffer directly to the receiver's buffer using only a single `memcpy()`. However, we have found faster ways to transfer data between the buffers for small messages and large message sizes (`memcpy()` remains ideal for medium-sized messages).

For messages smaller than 256 bytes, we have found that a two-copy scheme is actually ideal. MPI communication consists of two phases: message matching and data transfer. One cache miss is incurred during message matching when the receiver reads the matching information from the sender. A second cache miss is incurred when the receiver begins transferring the message data. The second cache miss can be avoided by having the sender copy its data to an in-line buffer located contiguously in memory after the matching information. Thus when the receiver matches a message, the associated data is prefetched by the processor and the second cache miss is avoided.

We also observed that a single core cannot fully utilize available memory or cache bandwidth using `memcpy()`. For messages larger than 4-12kB, we involve both the sender and receiver in transferring data between the send and receive buffers. The message is broken into a number of blocks. When a receiver matches a message, it sets a flag for the sender and begins copying data one block at a time, incrementing an atomic offset pointer to the next uncopied block. If the sender sees the flag is set, it also begins copying blocks and incrementing the offset pointer. The benefit of this technique is higher bandwidth utilization for larger messages. Another benefit is dynamic communication/computation overlap. If the sender has other work to do, the receiver proceeds on its own. Otherwise, the sender helps accelerate the communication it is waiting to complete.

1.3. Ownership Passing

Up to this point, we have discussed optimizations that speed up MPI transparently to applications by making better use of shared memory. However, overhead is still incurred when passing messages due to data copy operations. Instead, we can pass ownership, or control of, the communication buffers. Unlike previous message passing protocols that claim to be *zero-copy*, ownership passing never touches the data being communicated (*zero-touch*). Communication overhead is reduced to that of the message matching phase, which is constant regardless of message size.

Ownership passing reinterprets the concept of distributed memory in a way that retains simplicity while taking advantage of shared memory. Traditional distributed memory partitions the application's address space into static private memory blocks. In contrast, ownership passing allows this partition to be dynamic, with memory regions moving from one private space to another. The resulting flexibility makes it possible for message passing applications to utilize shared memory hardware in ways similar to native shared memory applications but without concern about data races and other complications of the shared memory model.

We have implemented our ownership passing technique as an extension to MPI. `OPI_Give()` and `OPI_Take()` operations analogous to `MPI_Send` and `MPI_Receive`, respectively, are used to pass ownership of a buffer from one MPI rank to another. Ownership passing naturally leads to a producer-consumer pattern in which many buffers are allocated, filled, consumed, and released. Our `OPI_Alloc()` and `OPI_Free()` routines implement a memory pooling technique to reduce the overhead of many `malloc()`/`free()` calls, and to improve buffer locality by returning released buffers to their original rank without introducing additional synchronization overhead.

1.4. Shared Memory Techniques

HMPI and ownership passing are just the first steps in leveraging shared memory hardware with MPI . We can further optimize MPI applications by taking advantage of the shared heap directly in applications. In particular, existing shared memory synchronization primitives are functional across processes in the shared heap environment. Mutexes, semaphores, signals, and even basic atomic operations like compare-and-swap or fetch-and-add work as expected. We will show how shared memory synchronization primitives can replace explicit MPI communication calls such and send and receive.

Consider a common communication pattern in which the sender packs sparse (non-contiguous) data into a buffer and sends it. Another MPI rank receives the buffer and unpacks the data into non-contiguous locations. The pack and unpack stages are written

as simple for-loops on the sender and receiver. Our shared memory heap makes it possible to perform code motion and loop fusion operations on the pack and unpack loops. Since both the sender and receiver share the same address space, we can move the unpack loop code from the receiver to the sender. The array assignments in the pack and unpack loops can be aligned, allowing the statements in the two loops to be *fused*. The result is a single loop that moves data from its non-contiguous locations on the sender directly to its non-contiguous locations on the receiver. When combined with shared memory synchronization primitives, all MPI communication is completely eliminated. In addition, loop fusion eliminates the need for intermediate communication buffers.

Loop fusion is more specific than ownership passing (i.e., mainly pack-communicate-unpack patterns), but offers greater performance gains. Pointers to data structures can be exchanged among MPI processes once (or occasionally as data structures evolve), and then used many times by fused loops that never make explicit MPI calls for communication. This approach leads to ideal use of shared memory for communication performance, but from the perspective of a distributed memory programming model.

1.5. Contributions

In this dissertation, we introduce a user-space shared memory heap (that requires no special operating system support) and then use it to develop novel techniques which better utilize shared memory in the context of distributed memory MPI . Hybrid MPI (HMPI) implements our new techniques in a portable library that works transparently with existing applications and MPI libraries (Chapter 2). In doing so, we have reduced communication overheads and made it possible to directly share memory in a distributed memory programming model, thus addressing its perceived limitations.

Our shared memory heap transparently enables single-copy intra-node communication via MPI for higher bandwidth and lower latency than existing two-copy communication protocols. In addition, we present two protocols that further increase performance for small and large messages beyond the speed of a single copy operation. Both protocols accelerate applications without modifications (Chapter 3).

We present two techniques for extending MPI to take further advantage of a shared memory heap. Ownership passing (Chapter 4) enables communication without copying data *zero-touch*. Loop fusion (Chapter 5) builds on ownership passing by moving code from one MPI rank to another, eliminating serialization loops and communication altogether.

In this work we present new optimizations and apply them by hand to demonstrate their effectiveness. Although not the focus of this dissertation, collaboration has resulted in development of compiler transformation techniques that automatically apply ownership passing and loop fusion optimizations to existing applications without additional developer effort [**7**, **23**–**25**].

We have published conference paper on the Hybrid MPI library, shared memory heap, and message passing protocols. [**23**]. Ownership passing utilizing an earlier thread-based version of HMPI was published in another conference paper. [**24**]. Early work on loop fusion has been released as a LLNL technical report. [**7**].

In the future, we expect that creative use of ownership passing and the shared heap will lead to a wide range of optimizations that eliminate the communication overheads associated with a distributed memory programming model, and MPI in particular.

2

Hybrid MPI

With the end of processor frequency scaling, performance and efficiency improvements in processor designs are achieved primarily by increasing the number of cores on a processing chip. The most common type of architecture for these designs is based on fully-featured compute cores connected via coherent shared memory, which provides significantly higher application developer productivity than alternative, more constrained designs. MPI is the de facto programming model for large-scale computing, used to implement the vast majority of scalable scientific applications. However, it was originally designed for systems where single-core compute nodes were connected by an inter-node network. Even as the MPI standard and individual MPI implementations have worked to adapt to new types of systems, the poor support (as demonstrated throughout this

dissertation) MPI implementations provide for many-core shared memory architectures has forced developers to use alternative programming models such as OpenMP [**71**] or OpenCL [**67**] to parallelize computations on such hardware, using MPI only for inter-node communication.

Despite the limitations of today's MPI implementations, the MPI programming model is highly compatible with the needs of applications. Indeed, a large body of MPI-based applications exist today, representing a significant investment of time and effort.

MPI simplifies parallel programming by defaulting memory to be private to each execution context and requiring the developer to explicitly indicate any communication. In fact, this expression of parallelism (via explicit communication) has benefits for shared memory, as it encourages data locality. We strive to leverage prior work based on MPI rather than discarding it (rewriting a code altogether) or ignoring it (modifying for an MPI+X approach). The challenge for MPI implementations is to provide developers this efficient abstraction across a wide range of architectures, including those where memory is fully shared or where only restricted communication primitives are available.

This chapter focuses on the design of MPI libraries for many-core processors connected via shared memory hardware. Given the wide variety of applications that use MPI and systems on which they run, our goal is to ensure that peak shared memory communication performance is available to these applications without sacrificing (i) portability, (ii) inter- and intra-node communication performance, and (iii) with no need for administrative access to modify the system. We present and evaluate the design of a new MPI library called Hybrid Message Passing Interface (HMPI) that is optimized for intra-node shared memory communication. HMPI composes with traditional MPI libraries optimized for inter-node communication by using only standardized MPI operations to inter-operate with them. Composing HMPI for intra-node communication with a traditional MPI for inter-node communication produces a comprehensive communication system for clusters of shared-memory nodes. In later chapters, we demonstrate this experimentally by composing HMPI with MVAPICH2 [**57**] and Blue Gene/Q MPI [**62**].

The intuition of our design is that efficient use of shared memory hardware requires the memories of MPI ranks running on the same node (shared memory domain) to be shared with each other (in MPI terminology a 'rank' is an execution context that may be an OS process, thread or some other entity). Fundamentally, sharing memory among MPI ranks allows MPI applications to utilize shared memory hardware as efficiently as threaded applications, making it possible for developers to achieve high performance on modern architectures without significantly changing their applications.

The key contributions of this chapter are the following:

(1) A simple, highly portable layered library design facilitating MPI communication research.
(2) An analysis of thread-based MPI design and identification of its limitations.
(3) A shared memory allocator technique for transparently enabling shared memory between local MPI ranks without modifying application code.

In the next section, we discuss our HMPI layered library in detail. Section 2.2 discusses the process-oriented rank design currently used in most MPI libraries, and why it is not sufficient for our shared memory research. The alternative is a thread-oriented rank design, which has the necessary shared memory functionality, but has severe drawbacks that are discussed in Section 2.3. Finally, Sections 2.4 and 2.4.1 discuss the process-oriented shared heap approach we have taken in HMPI .

2.1. Hybrid MPI

We have implemented a Hybrid Message Passing Interface (HMPI) library to investigate shared memory (intra-node) message passing techniques. The term 'hybrid' comes from the fact that two memory models are combined in one design. Rather than building an entire MPI implementation from the ground up, we have taken the approach of layering HMPI on top of any existing MPI library. Figure 2.1 illustrates how HMPI layers in between applications and MPI. There are two advantages to our approach: portability and transparency.

Portability: HMPI works on top of any existing MPI library simply by linking it into the application. We are able to extend both open- and closed-source MPI implementations on multiple platforms.

Transparency: No source code transformations, object file or library modifications are needed. Neither the application nor the underlying MPI or memory allocator library need to be changed or made aware of HMPI's presence.

The combination of these advantages allows us to experiment with new message passing techniques on multiple platforms (including those with closed-source, proprietary MPI implementations) with minimal effort. No system modifications are required, nor any administrative privileges. The entirety of our work was developed and evaluated on production HPC systems with regular user accounts. As a result, others can easily reproduce and extend our work on their own systems.

PACX-MPI [**27**, **43**] is an earlier layered-library MPI design for grid systems. In their case, they implemented cross-cluster MPI communication in their library and relied on the native MPI libraries for communication within each separate cluster. Like our work, their motivation is achieving performance productivity by leveraging existing work on platform-optimized MPI libraries. Instead of inter-cluster communication, we focus on optimizing intra-node communication and providing extensions to MPI for further leveraging shared memory performance. In general, we believe the layered MPI library design is useful for researching new communication techniques without losing portability. For example, it would also be useful for heterogeneous systems containing GPUs, coprocessors, and other accelerators.

HMPI's layered library design is a key contribution of this dissertation that forms the sandbox in which we can perform further research. Specifically, we focus on point-to-point communication techniques in HMPI, including faster communication protocols (Chapter 3) and extensions to expose shared memory to applications (Chapters 4 and 5). We intercept messages destined for ranks within the same node and optimize their communication while passing inter-node messages through to the underlying MPI layer. All other MPI functionality is passed through in the same manner. MPI libraries are allowed to

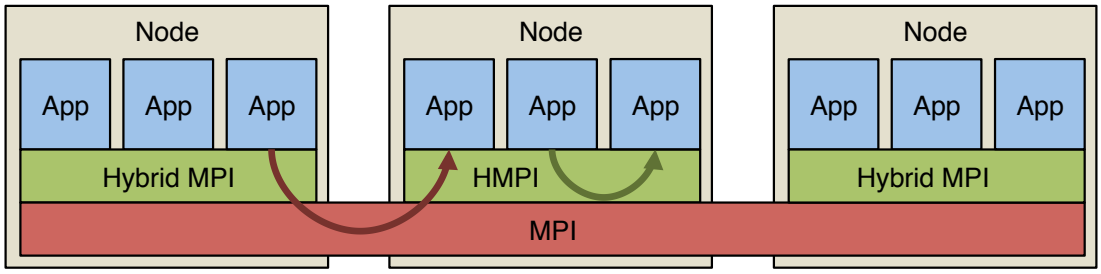


FIGURE 2.1. Hybrid MPI links between applications and an existing MPI library. Intra-node communication is handled by HMPI (green arrow), while an MPI is used across nodes (red arrow).

implement the collective communication routines in terms of the point-to-point routines. However, the manner in which HMPI is linked into the software stack does not cause point-to-point communication calls within the underlying MPI to invoke HMPI. S. Li et al. have researched NUMA-aware collective communication algorithms using HMPI and shared memory [**56**].

To provide some background, Section 2.2 discusses the traditional processed-based, distributed memory design used in practice by most MPI implementations. However, we desire a new design that more tightly integrates shared memory into MPI to facilitate our research on shared memory passing techniques. Previous work has investigated the idea of a thread-based MPI design in which each rank is a thread sharing memory with all ranks on the same node. Indeed, we initially adopted this approach in HMPI. For reasons we will discuss in Section 2.3, this approach is not prevalent in practice due to performance limitations and required application code modifications. We utilize a third approach, discussed in Section 2.4, that assigns each rank to its own process but shares heap memory among all ranks in a node. Our approach unifies the benefits of process-based and thread-based MPI design without the drawbacks of either approach.

2.2. Process-based MPI

Although the MPI standard [**65**, **66**] does not prescribe how MPI ranks are implemented, the traditional assumption has been that each rank is an OS process with its own private memory. Figure 2.3 illustrates this layout. Normally every rank (process)

```
int main(int argc, char** argv)
{
    int size;
    int rank;
    int data = time(NULL);

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(size > 0 && rank == 0) {
        MPI_Send(&data, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if(size > 0 && rank == 0) {
        MPI_Recv(&data, 1, MPI_INT,
                0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    MPI_Finalize();
}
```

FIGURE 2.2. Simple example of an MPI program.

is an instance of the same program; this approach is known as Single Program Multiple Data (SPMD). Each rank executes the same code (single program), but works on its own portion of the problem (multiple data). A trivial "six function" MPI program is shown in Figure 2.2. The program initializes, determine its ranks and the total number of the ranks, then sends and receives some data.

The process-based MPI design pre-dates multi-core technology, originating from an era when single-processor, single-core systems were the norm. Thus, intra-node communication was far less important. Today, the advantage of the process-based design is that it makes it easy to coordinate inter-node communication by multiple cores. Since each core is used by a separate process, their MPI libraries maintain separate state and thus require no synchronization. Network interfaces are typically designed to provide each process a separate context in which to coordinate its outgoing and incoming communication, so no MPI-level synchronization is required to access the network either.

The limitation of this design is in communicating among different ranks that are executing within the same shared memory node. Now ubiquitous multi-core processors are

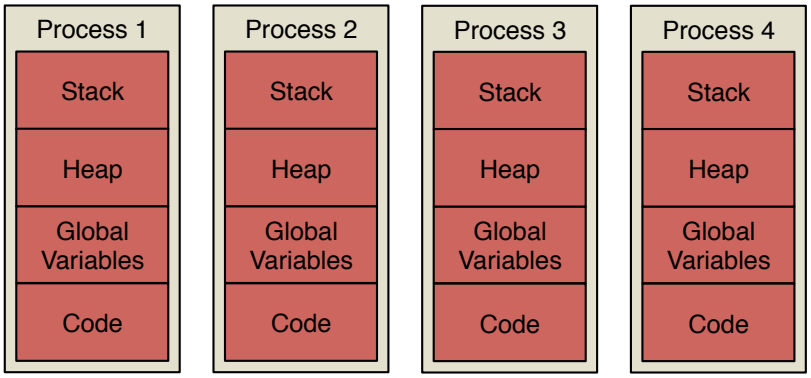


FIGURE 2.3. Memory layout in the traditional process-based MPI design. No application-visible memory is shared when MPI ranks are processes.

now making this limitation far more important. Within a node, MPI libraries communicate via relatively small shared memory regions mapped pair-wise between ranks [**9**, **13**]. This design already has an issue: with the growing number cores per node, any sort of pair-wise technique is not a scalable solution. Resource usage grows as the square of the number of MPI ranks per node, which is often the number of cores per node.

To transfer a message via shared memory, the sender copies data into the region shared with the receiver. Next, the receiver copies out of the region and into the application's receive buffer. The shared memory regions act as a bounce buffer, and two copies are required per message. As a result, precious memory bandwidth is wasted copying the same data multiple times. A common optimization for large messages is to overlap and pipeline the two copies by breaking the message into blocks, allowing the sender and receiver to perform their respective copies simultaneously. However, this pipelined approach limits its communication-computation overlap, since both ranks are dependent on each other to perform their respective copy operations.

As a related work: there is an alternative to using two-copy protocols to communicate between ranks within a node. A network interface can be used as a loop-back device [**9**]. In this approach, ranks communicate using the same network interface and protocols regardless of whether the peer rank is local or remote. The advantage is that the MPI implementation is less complex, and depending on the network, data transfer occurs asynchronously

rather than tying up the processor (improved communication-computation overlap). The downside is that network communication has higher overhead, most acutely felt as higher small message latency. Typically, this approach is only used on systems with high-end networks and/or fast network co-processors, while the two-copy method described above is used on commodity clusters.

2.3. Thread-based MPI

The limited memory sharing ability of process-oriented MPI implementations has motivated research on implementations where MPI ranks are implemented as OS threads, all of which execute within the same process [**20**, **37**, **41**, **58**, **74**, **83**]. Figure 2.4 illustrates this layout. The MPI interface and SPMD approach remains largely the same; for example the simple program in Figure 2.2 works as-is with a thread-based MPI. Threads are an excellent choice since they share all their memory by default—this is exactly what we want for research on using shared memory to its fullest in MPI. However, many MPI applications are written with the assumption that global variables are private to each MPI rank. While threading gives each rank its own stack and heap within the shared address space, one set of global variables is shared among all MPI ranks in a node. Application state becomes corrupted as different MPI ranks write to the common global variables, which may exist within the application and in any libraries they link with.

Developers of thread-oriented MPI implementations have attempted to resolve this problem in two ways. First, they have developed techniques to privatize global variables so that each thread is provided its own copy. At the source code level, privatization can be done using thread-local storage, using the `__thread` keyword available in many C compilers, or using compiler transformation tools [**20**, **69**, **74**]. There has been work on tools that modify object files to privatize global variables when the source code is not available [**69**].

Given the complexity of privatization, especially in library code, an alternative approach is to adjust the use of libraries to ensure that no global variables are used. This involves replacing regular library calls with their thread-safe variants, for example using `strtok_r()` instead of `strtok()`. Where thread-safe alternatives are not available (e.g.,

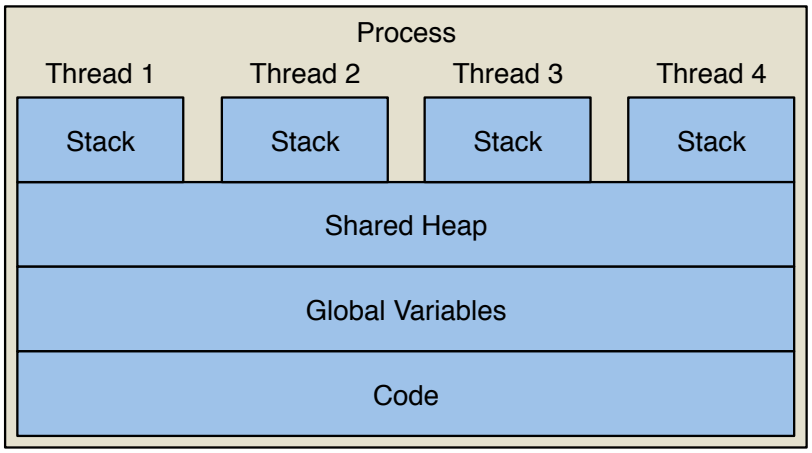


FIGURE 2.4. Memory layout in the thread-based MPI design. All application visible memory is shared when MPI ranks are threads.

the `getopt()` function uses static variables internally) locks are required to protect access to the function. While it is possible to build compiler tools to perform this replacement, they would require knowledge about each library and its thread safety guarantees.

2.3.1. Network Performance. Where high performance is desired, MPI implementations must use a network interface directly. Depending on the network, issues can arise due to the use of multiple threads. In particular, not all network interfaces provide thread safe APIs. Any MPI using multiple threads must protect all inter-node communication using a lock. Process-based MPI implementations face no such requirement. Unfortunately, an MPI-level network lock results in high contention for network resources and reduced performance. Figure 2.5 shows the effect of this contention during communication portions of the MiniMD application. We measured the time taken by the communication portions of MiniMD using varying numbers of ranks on 16 nodes of a Sandy Bridge x86 cluster[1]. The Sandy Bridge system has an InfiniBand-based Performance Scaled Messages (PSM) network that is not thread-safe.[2] Network resource contention will be a problem when using any network that does not support multiple threads as efficiently as multiple processes.

[1]See Section 3.4 for details on the MiniMD application and Sandy Bridge system.
[2]The OpenFabrics software distribution for InfiniBand networks supports multi-threading.

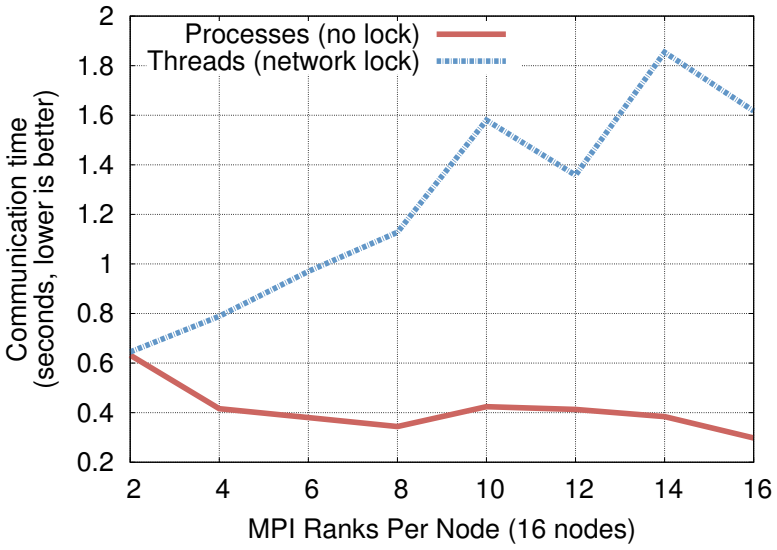


FIGURE 2.5. Affect of overhead due to lock contention for network resources during the communication phase of the MiniMD application with strong scaling. However, when using threads, the communication overhead increases due to lock contention.

Driver thread-safety also creates issues when using multiple threads with a process-based MPI implementation. If MPI is running in `MPI_THREAD_SINGLE`[3] mode, application threads must use locking to control access to MPI, which creates contention. However, if MPI provides `MPI_THREAD_MULTIPLE`[4] mode, then the library's calls to the network drivers from the various threads must also be locked inside MPI because the drivers are not thread-safe. This is also one of the reasons why multi-threaded MPI applications are not common and perform sub-optimally [**84**]. Work on utilizing additional threads inside an MPI library for asynchronous communication progress [**35**] also found synchronization to be a challenge.

We have made the following observations on thread-based MPI design:

- Threads share all memory by default, which meets our design criteria for tighter shared memory integration.

[3] `MPI_THREAD_SINGLE` is the default mode in which MPI supports only one thread per rank, but in practice provides the best performance.

[4] `MPI_THREAD_MULTIPLE` mode enables full thread safety in MPI, enabling applications to use multiple threads without wrapping MPI communication in critical sections. However, MPI's performance may be reduced due to internal synchronization.

- Global and static variables are shared among ranks, breaking the assumption of a private address space and requiring the vast majority of MPI applications to be modified.
- Tools exist to privatize global variables, but ensuring thread-safety of all libraries, including the standard C library, is very hard to solve in practice.
- Multi-threaded safety requirements take the 'performance' out of high-performance networks.

Although thread-based MPI implementations provide the level of shared memory integration needed to support all the optimizations shown in this dissertation, the remaining observations suggest that thread-based MPI implementations are not a practical approach. We note that no thread-based MPI has been widely adopted in practice, and conclude that the same holds true for the future of multi- and many-core HPC systems.

2.4. Shared Memory In Processes

The goal of our work is to develop an implementation of MPI that is (i) optimized for shared memory hardware, (ii) works on existing operating systems with no root access, (iii) is compatible with any inter-node MPI implementation and (iv) provides peak performance for both intra- and inter-node communication. Given the challenges faced by MPI implementations that use threads to implement MPI ranks, we have chosen to implement them using OS processes. However, we must still develop some mechanism to more effectively share memory among processes.

XPMEM [**72**,**88**] is a Linux kernel extension that allows one process to map the memory of another process into its own address space. SMARTMAP [**5**,**6**] is a similar extension for the Catamount lightweight kernel used on Cray systems. Using XPMEM or SMARTMAP, each MPI rank could map the entire address space of every other rank on the node. However, it would not be possible to map the same memory at the same address on every rank, so address translation is needed. A solution similar to our shared heap allocator (Section 2.4.1) could set up a large heap region on one process, then share and map it at

the same address on every rank. However, the downside of XPMEM is that it is not installed by default on standard Linux systems; SMARTMEM is only found on one specific non-Linux OS. Using these extensions requires a specialized system, system modifications, and/or administrative privileges.

Blue Gene/Q systems have the ability to disable the virtual memory abstraction, removing the protection barriers between processes. This feature is enabled by setting an environment variable (`BG_MAPCOMMONHEAP`) **[62]**. All processes (ranks) on the compute node execute in one shared address space; the OS takes care of mapping each process' segments to unique locations. This is exactly the kind of environment we desire for shared memory optimizations; all shared memory is shared at the same address by default without any extra engineering on our part. However, IBM implemented this feature to maximize the available memory; it is intended as a work around for limitations in how they divide memory among processes. Although the common heap feature currently works well for message passing via HMPI, we ran into severe problems when experimenting with ownership passing (Chapter 4). After consulting with LLNL and IBM system experts about these issues, we concluded that the feature isn't suitable for use with HMPI. In particular, we would not be using the common heap feature as it was intended, and IBM makes no guarantee that our use case should work now or in the future. Indeed, they were impressed that we were able to get anything working at all.

On the systems we use for experimental analysis in this dissertation, including Blue Gene/Q, we utilize the user-space shared heap allocator discussed in the following section.

2.4.1. Shared Memory Heap Allocator. For systems without shared heap functionality built in, we have developed a replacement for the default memory allocator that shares heap memory among all processes in a similar manner. Our shared memory heap allocator enables the same shared-memory techniques on POSIX-compliant (e.g., Linux) systems without requiring installation of kernel extensions, modification of system libraries, or administrative permissions. Our shared heap allocator is stand-alone and not MPI specific;

we imagine it is also useful for other forms of shared memory communication such as PGAS run-times or even OpenMP or OpenCL adapted across processes.

First, some background on memory allocator design. Normally, the memory allocator incrementally requests memory from the operating system using the `sbrk()` or `mmap()` system calls. Small allocation requests ('small' may be defined differently depending on the allocator implementation, but typically less than one page, or 4kB[5]) are serviced using a pool of memory acquired using `sbrk()`. When these small allocations are freed, they are marked as available for reuse by the allocator rather than actually returning the memory to the system. Larger allocations are serviced using `mmap()`. `Mmap()` is not used for small messages due to system call processing overhead and because `mmap()` rounds up to a multiple of the page size (4kB), leading to wasted memory when there are many small allocations. Memory allocated using `mmap()` is normally returned to the system using `munmap()`. The `mremap()` routine allows memory allocated using `mmap()` to be moved to a larger region by changing the virtual memory mapping rather than copying, which is a useful optimization for `realloc()`. `Calloc()` can take advantage of the fact that the kernel initializes pages backing a mapped region using copy-on-write from a zero-initialized page, avoiding the need to write zeros to the new allocation.

In order to make our system fully transparent, we override the system's default memory allocator to allocate memory from a specially crafted shared memory pool. Using this approach, we have modified both Doug Lea's malloc library (dlmalloc) **[52]** and Google's tcmalloc library **[40]** to transparently provide shared memory from `malloc()` and related routines.

To provide memory for a shared heap, we allocate and map a large shared memory region (larger than physical memory) to the same address on each MPI rank. There are several tools available for doing this; we prefer using `mmap()` with a file located on a ram disk. We also support the use of SYSV shared memory segments. Both provide the necessary functionality, but due to varying system configurations (e.g., maximum shared memory limits), one may be more practical than the other.

[5]Binary units are used exclusively in this dissertation: kB = 2^{10} bytes, mB = 2^{20} bytes, gB = 2^{30} bytes, etc.

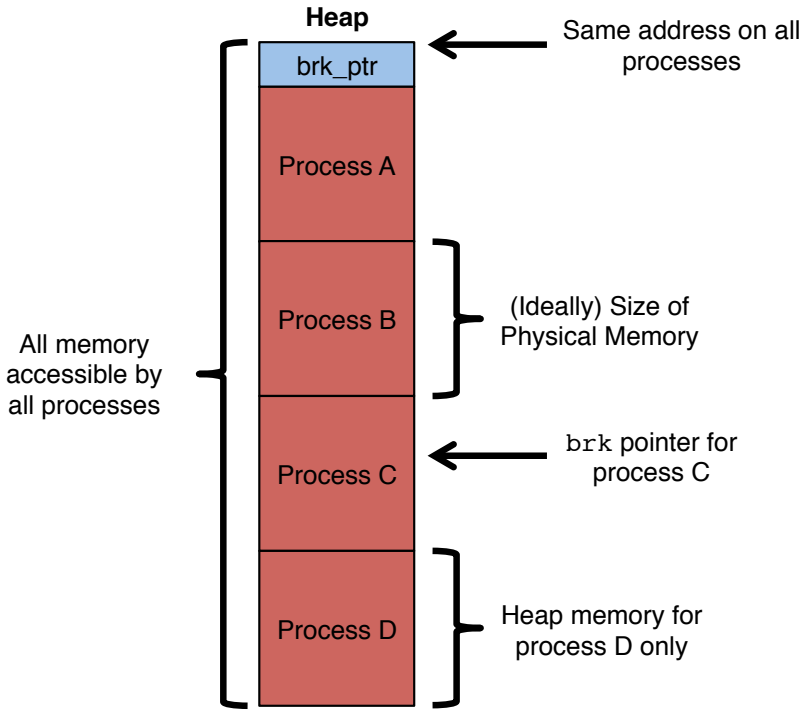


FIGURE 2.6. Illustration of the shared memory heap region. Each process maps the entire region at the same address, then allocates its memory from only from its portion.

The shared region is divided evenly among the ranks on the node, and each rank allocates memory only from its part of the mapped region. This approach eliminates the need for any synchronization between processes within the memory allocator, but still allows for sharing of memory. On the Sandy Bridge system with 32 gB RAM per node, we were able to reserve space for up to 32 gB per MPI rank, for a total of $32\,gB \cdot 16\,ranks = 512\,gB$. Without swap, total memory usage cannot exceed physical memory, but this larger shared region allows for unbalanced memory usage across the ranks—up to 32 gB by one rank.

We implement our own version of `sbrk()` that requests memory from a shared memory region mapped on all MPI processes. The design is simple: we maintain a *break pointer* for each process, which points to the end of the allocated portion of that process' shared region. `sbrk()` adjusts that pointer to allocate or free memory; the memory allocator takes care of piecing allocated memory out to the application. Figure 2.6 illustrates the layout of the shared memory heap. `mmap()` is more difficult due to how memory allocators use it

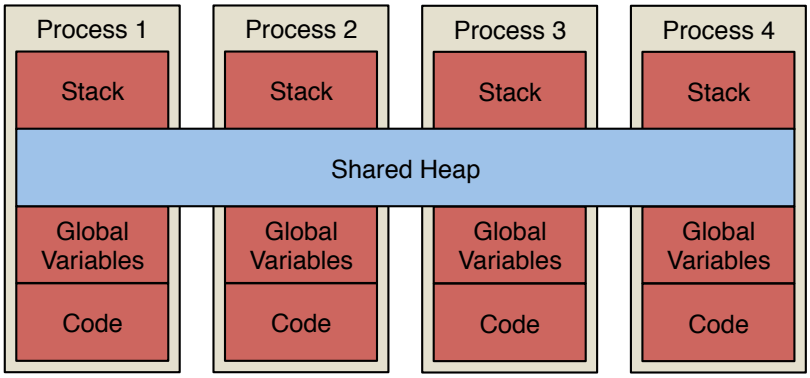


FIGURE 2.7. Memory layout of processes with our shared heap allocator. Dark red segments are private to each process, while the light blue heap segment is shared among all processes in a node.

once for one allocation, then free the region. Such behavior results in fragmentation, which in turn effectively requires a memory allocator to be written in order to fake `mmap()` using one large shared region and without any further OS support. Thus, when we modify the system allocators, we configure them to only use `sbrk()` to acquire memory regardless of allocation size. The downside is that we miss out on the `realloc()` and `calloc()` optimizations discussed above. Nonetheless, we have developed a shared heap allocator that works entirely in user space so that we can experiment on platforms without such functionality built in.

Figure 2.7 illustrates how our shared memory allocator connects multiple processes together. Stack, global variables, and code are private to each process, but the heap is shared. Our memory allocator provides the same shared-heap benefits as thread-based MPI and systems with kernel extensions. However, we incur none of the global variable privatization challenges encountered by thread-based MPIs and do not rely on specific operating systems, resulting in maximum portability. Our approach works on any platform that allows shared memory, allows overriding memory allocation calls (e.g., via weak symbols), and provides an MPI library.

In addition to the heap, MPI allows communication buffers located in global variables and the stack segment. Section 2.3 discussed why sharing global variables is problematic and undesirable. Sharing stack memory would not cause problems, but there is no good

mechanism for doing so. If the application's `main()` routine only operates on local variables before calling a routine we control (i.e., `malloc()` or `MPI_Init()`), it is possible to use the `swapcontext()` et al. routines to switch to a stack located in shared memory. More generally, a compiler tool could transform the application's source code or object files to enable sharing of stack memory. Since the benchmarks and applications we have considered so far primarily communicate using heap memory, we have not implemented any form of shared-memory stack. Chapter 3 discusses how HMPI handles communication when user buffers are not located in shared memory.

2.5. Summary

In this chapter, we introduced our Hybrid MPI library, which will serve as the platform for shared memory message passing research. HMPI uses a layered-library design, linking in between an application and an existing MPI library. It is portable: no kernel extensions, system library changes, or administrative access is required. No modifications to applications or other MPI implementations are required, making it transparent to the existing software ecosystem.

To meet our shared memory requirements, we evaluated the process-based design of existing MPI libraries and found it insufficient for our purposes; shared memory is merely used as a sort of bounce-buffer for data transfer. Thread-based MPI designs provide the shared memory we need, but we showed that such a design is impractical due to thread safety and performance reasons. Our final design for HMPI incorporates a shared memory heap into the existing process-based MPI design, yielding shared memory without the drawbacks of a thread-based approach. We utilize shared heap features on systems that have it (e.g., Blue Gene/Q), and designed a user-space shared heap memory allocator for systems that do not, for example commodity clusters.

In the following chapters, we will demonstrate multiple ways in which shared memory can be effectively used with MPI for better performance. Chapter 3 discusses new protocols for faster MPI communication. Chapter 4 introduces Ownership Passing to MPI, a small MPI extension which eliminates message copying overhead and exposes shared

memory to the application. In Chapter 5, we take advantage of shared memory to completely eliminate MPI communication by fusing pack and unpack loops between ranks.

3

Shared Memory Message Passing

MPI was designed when single-processor nodes connected by a network were the norm for clusters. Its distributed memory model fits this type of system perfectly: each processor has exclusive access to all of its local memory and always communicates via the network. Today, multi-core processors are ubiquitous and it is common to have 16, 64, or more cores sharing one memory domain in a single node. Depending on the application's communication pattern, a substantial portion of communication now occurs among MPI ranks within a node. Each node normally has one network interface, which is now shared by multiple processor cores (and ranks). As a result, communication performance within a node has become just as important as network performance.

MPI implementations use a generic network abstraction through which various network transports can be supported in a modular and portable manner, isolating network-specific details from a core message passing architecture [**9**, **26**]. Such a design makes limited use of shared memory, effectively communicating through a small pipe in a manner similar to a network. In particular, a pipelined two-copy protocol using a small shared buffer is used [**9**, **13**]. There are several disadvantages to this approach. Both the sender and receiver must copy the message in its entirety at the same time. Although pipelining the two copy operations leads to transfer rates approaching that of a single copy, moving the data through two different processor cores pollutes more cache memory than necessary. Finally, performing two copies creates twice as much memory traffic as is necessary to move data from one location to another.

Prior work has explored techniques for transferring data using a single copy via kernel extensions: kcopy [**85**], KNEM [**28**, **60**], and LIMIC [**89**]. The advantage is that data is transferred using a single copy rather than one. However, special kernel extensions must be loaded, and are not installed by default on Linux-based operating systems. Furthermore, these extensions require performing a system call to initiate data transfer, which has too much overhead to provide performance gains for small messages.

XPMEM [**72**, **88**] and SMARTMAP [**5**, **6**] allow memory from one process to be directly mapped into another. These kernel extensions enable single-copy data transfer, and are sufficient to implement the further protocol optimizations we present in this chapter. However, XPMEM and SMARTMAP are kernel extensions, and have the same availability limitations as kcopy, KNEM, and LIMIC.

In this chapter, we discuss this related background work, and then present the communication mechanisms implemented in Hybrid Message Passing Interface. MPI communication has two basic stages: message matching and data transfer. Section 3.1 outlines MPI's message matching rules, and describes our design implemented in HMPI. In Section 3.2 we discuss communication protocols for transferring data in MPI. After describing the current state of the art and related work, we discuss and show experimental results for the protocols implemented in HMPI. First, we utilize shared heaps to perform single-copy

message transfers entirely in user space and without requiring any kernel extensions or other system modifications. Second, we have developed two new protocols that improve on single-copy performance for small and large messages. In Section 3.4 we show experimental results for two applications.

The key contributions of this chapter are the following:

(1) A discussion of MPI communication phases and their design using shared memory in HMPI.

(2) Two new point-to-point protocols for message passing that utilize a shared address space for better performance than a single-copy protocol.

(3) Analysis of shared-memory message passing performance on an x86 Sandy Bridge system using our shared memory allocator technique, and on Blue Gene/Q, a system providing a shared address space feature.

3.1. Message Matching

In MPI, a message is created by a send operation and then passed, or transferred, to the receiver. Since multiple messages may be in flight between any pair of ranks, a matching process is required to determine which send operation matches which receive operation. Each message has four associated parameters that are used to determine how it is matched:

- Receiving Rank
- Sending Rank
- Tag
- Communicator

Matching can be carried out by either the sender or the receiver, but is wildcard matching is easier when done by the receiver. The receiving (destination) rank is specified by the send operation. Likewise, the sending (source) rank is specified by the receiver. Receivers may specify `MPI_ANY_SOURCE` to indicate that a message from any source rank is a valid match (sends have no wildcard values). The tag is a value specified by the application that must match on both the send and receive operations. Receivers can specify `MPI_ANY_TAG` to indicate any tag value is a valid match. The communicator is an opaque object that

functions like a second tag in the context of message matching, but there is no wildcard value.[1] Excepting wildcards, the source, destination, tag, and communicator must match in order for a message to be transferred.

MPI specifies that messages are *non-overtaking*: if multiple messages with the same source, destination, tag, and communicator exist, they are matched in the order the operations were executed. In other words, the oldest outstanding receive operation must be matched with the oldest possible send operation. This ordering requirement makes message matching deterministic as long as `MPI_ANY_SOURCE` is not used. See the MPI standard [**65**, **66**] Section 3.5 for more details on ordering, progress, fairness, and resource limitations with respect to matching.

MPI libraries are free to implement matching however they wish as long as the above rules are met. Matching designs can very significantly depending on the system; in particular the features provided by underlying network interfaces can have a huge impact. For example, PSM networks have all the necessary support for matching messages in the hardware. On the other hand, a TCP/IP network requires an entirely software-based matching design.

An intuitive design for message matching in shared memory is to associate a message queue with each rank. Senders create a message object consisting of the above matching parameters and a few others things like the buffer location and datatype, then add it to the receiver's message queue. In addition to the message queue, each rank maintains a private list of outstanding receive operations. Inside MPI , there is a 'progress' routine that is called whenever the application calls an MPI routine. This routine attempts to make progress on the work of matching messages on the incoming queue to its receive operations, and is polled at a high rate when waiting for communication operations to complete. If a message is successfully matched, communication proceeds to the data transfer phase.

A converse design can be used where receive operations are queued on the associated sender. However, the `MPI_ANY_SOURCE` source rank wildcard is difficult to support: the

[1] Although communicators are opaque objects in MPI, they contain an identifier that is unique among all communicators in the running MPI application.

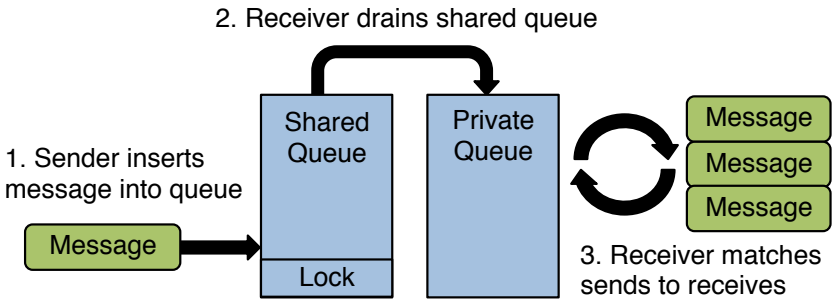


FIGURE 3.1. HMPI's matching design. Each receiver has two queues, one shared and one private. Senders insert messages into the shared queue protected by a lock. The receiver drains the shared queue into its private queue and enters a loop to match incoming sends to local receives.

receiving rank has no way of knowing which sender to queue its receive operation on. Any solution would require additional bookkeeping and/or synchronization, so a receiver-side queue is used in practice.

3.1.1. HMPI Matching Design. In HMPI, we implement two incoming message queues per receiver using linked lists. Figure 3.1 illustrates our design. One queue is globally accessible by all ranks. Senders add messages to the global queue owned by the rank for which the message is destined. Each global queue is protected by a Mellor-Crummey and Scott (MCS) lock [**63**]. An important benefit of the MCS lock is guaranteed First In, First Out (FIFO) ordering of lock acquisitions. When using a lock without this property (e.g., a simple compare and swap lock), some ranks could be blocked for long, unpredictable periods waiting to add a message to a receiver's queue. FIFO ordering ensures fairness and consistent application run times.

The second queue is private. When a receiver attempts to match incoming sends to local receives, it drains its global queue and appends incoming sends to its private queue. Since the queues are linked lists, the draining operation only involves updating the head and tail pointers. The receiver then attempts to match sends on its private queue to local receives. A second private queue enables the receiver to loop many times without need for synchronization, and ensures that messages cannot be matched out of order due to senders adding new messages to the queue while the receiver is in the middle of its matching loop.

Our dual queue technique minimizes contention between processes and is scalable since there is a separate lock for each receiver queue. The only scenario where contention increases as core counts increase is when many messages are sent to the same receiving rank. One solution would be to create multiple shared message queues per receiver and balance the load among them. This approach creates more synchronization work for the receiver, but reduces contention. We have not yet observed contention causing a bottleneck in HMPI, so we stick with one pair of queues per rank.

3.2. Communication Protocols

Once a send and receive operation are matched, the data transfer phase begins. MPI requires that data in a location specified by the send operation must be copied to a location specified by the receive operation, but does not specify how. In this dissertation and specifically in this section, we focus on methods for transferring data through shared memory. First, we discuss the shared memory communication protocols used in existing MPI libraries and prior work.

3.2.1. Existing MPI Protocols. A pipelined two-copy protocol is used for intra-node communication in existing MPI libraries [**9**, **13**]. Figure 3.2 illustrates this protocol. Shared memory regions are reserved between each pair of MPI ranks in a node. Data is transferred by breaking the message into blocks. The sender copies one block at a time into the memory region shared with the receiver, signaling to the receiver when the block copy is completed. The receiver then begins copying that block to the final receive buffer. Meanwhile, the sender can begin copying the next block to the shared region, and so on. Although message data is copied twice, performance approaches that of a single copy as the message size increases.

LiMIC [**89**] and KNEM [**28**, **60**] are OS kernel extensions adopted by MVAPICH [**57**], MPICH2 [**9**], and Open MPI [**26**] that provide a system call which can copy data from one process' address space to another with a single copy operation. Since the kernel has access to the private memory of every process on the system, it is able to copy data from one process's memory to another directly instead of relying on two copy scheme. KNEM

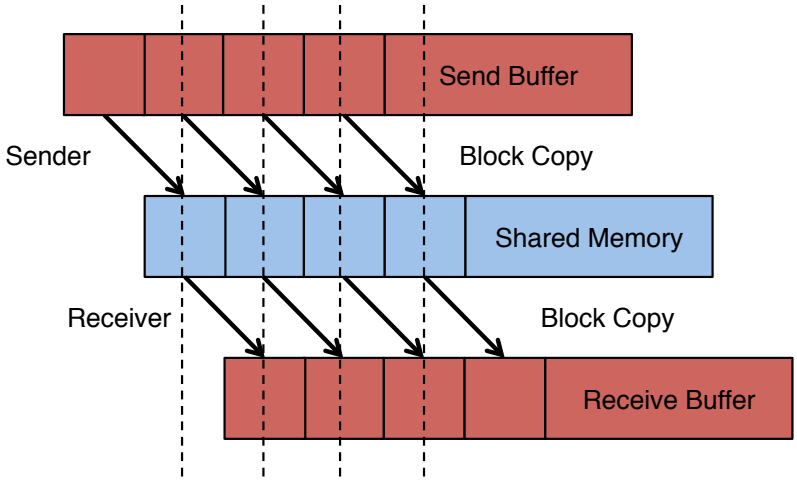


FIGURE 3.2. Pipelined two-copy protocol used by existing MPI libraries. The sender copies blocks into a shared memory region, notifying the receiver as each block copy completes. The receiver follows, copying each completed block out of shared memory.

also supports the Input/Output Acceleration Technology (I/O AT) feature of some Intel memory controllers. **[8, 30]** I/O AT is essentially a DMA engine that can be used to copy data asynchronously (independently of any processor). Since I/O AT offloads the copy operation, the processor's caches are not polluted by the data transfer.

The downside of LiMIC and KNEM is that they are kernel extensions, and are not part of standard Linux installations. Thus, they require manual modification of the OS kernel to be supported. I/O AT is a feature found only on certain Intel server-grade motherboards, so is not commonly available. Furthermore, these extensions require making a system call to transfer data. System calls require a time-intensive switch to privileged kernel mode, making these single-copy kernel extensions inefficient for small messages. Specifically, **[28]** Section 5.2 states that KNEM is not competitive for messages less than 16kB and **[89]** Section 5.1 indicates that LiMIC only provides an improvement for message sizes 4kB and up.

3.2.2. HMPI Communication Protocols. Although single-copy message transfer was our goal with HMPI , we have discovered that using `memcpy()` to transfer the data is often not the fastest method possible. We developed an 'immediate' protocol for small messages

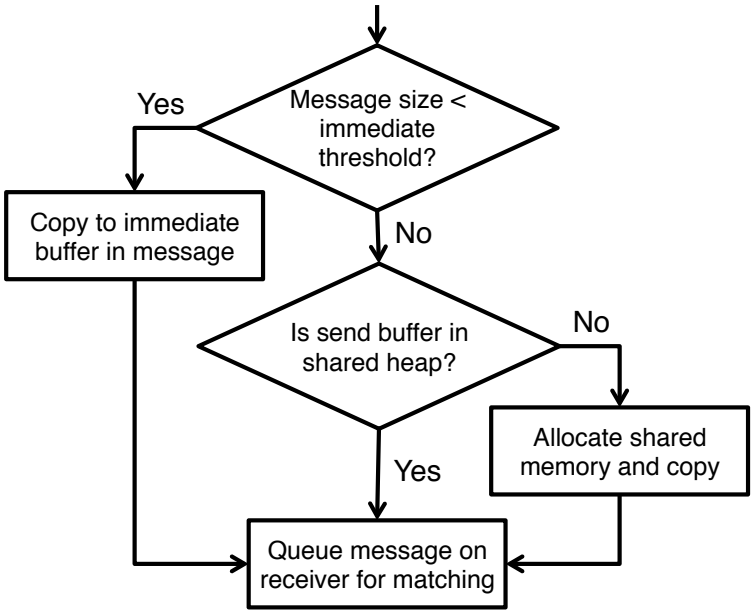


FIGURE 3.3. Sender protocol flow. The sender ensures its buffer is in the shared heap and uses the immediate transfer protocol for small messages.

less than the *immediate threshold* (currently 256 bytes, Section 3.2.3), and a 'synergistic' protocol for messages larger than the *synergistic threshold* (currently 4 or 16 kB, Section 3.2.4). We support buffers from global variables or the stack by checking the location of each buffer given to HMPI by the application. If the buffer address lies outside of our shared memory heap, we fall back to a two-copy transfer mechanism.

Before queuing a message, the sender goes through a series of checks as shown in Figure 3.3. If the message is small (i.e., <256 bytes), we enter the immediate protocol, inlining the message data with the message's matching information. If the application's send buffer is not located in the shared heap, we allocate a shared buffer and copy the data over. We ensure that send buffers, whether provided by the HMPI or the user, are always located in shared memory. Finally, the message is added to the receiver's shared queue for matching.

Once a message is matched, the receiver decides how to transfer data from the provided send buffer to the receive buffer. Figure 3.4 shows the decision process. If the receive buffer is not on the shared heap or if the message is too small to use the synergistic

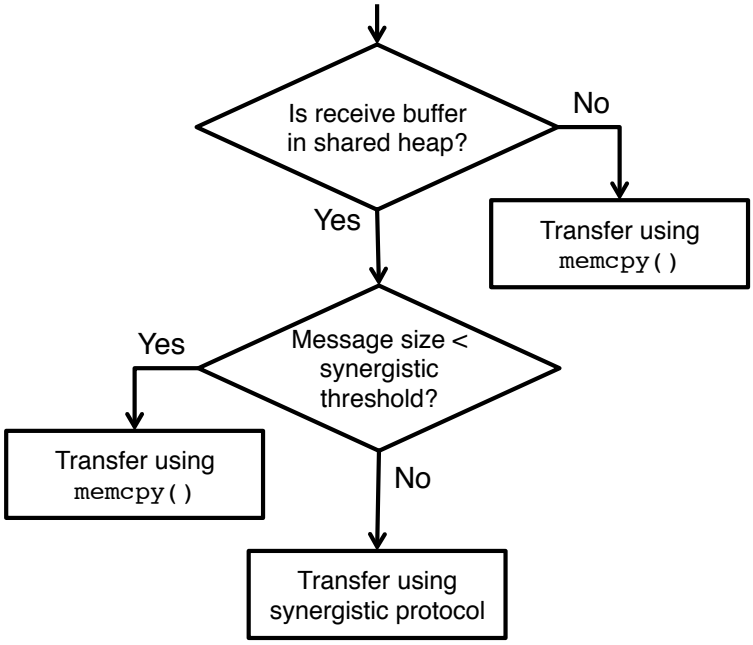


FIGURE 3.4. Receiver protocol flow. A single `memcpy()` is used if the receive buffer is not in the shared heap or if the message is too small for the synergistic protocol.

protocol, we use `memcpy()` to transfer the data (this case covers both single-copy and immediate protocols). For larger messages we enter the synergistic protocol.

3.2.3. Immediate Transfer Protocol. For small messages, the best latency is achieved by utilizing a two-copy method with the message data located immediately after the matching information (Figure 3.5). The performance advantage stems from the following simple observation: When a message is matched, the receiver accesses the source message information (source rank, tag, communicator), incurring a cache miss. With a single-copy data transfer approach, copying the message data will incur another cache miss, since that data has not been seen by the receiver. Inlining the message contiguously after the sender's matching information causes the hardware to bring the data into cache at the same time as the matching information, avoiding the second cache miss when copying the data.

As seen in Figure 3.3, the sender will perform the additional copy before queuing the message at the receiver. From the receiver's point of view, the immediate protocol is the same as single-copy transfer—just copy the data from the location provided by the sender.

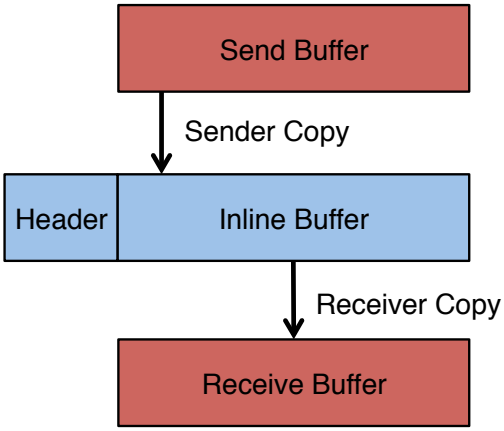


FIGURE 3.5. The immediate protocol. Senders copy their data to an inline buffer located contiguously after the message matching information. Once a receiver matches the message, it copies from the inline buffer to the receive buffer, avoiding a cache miss by exploiting data locality.

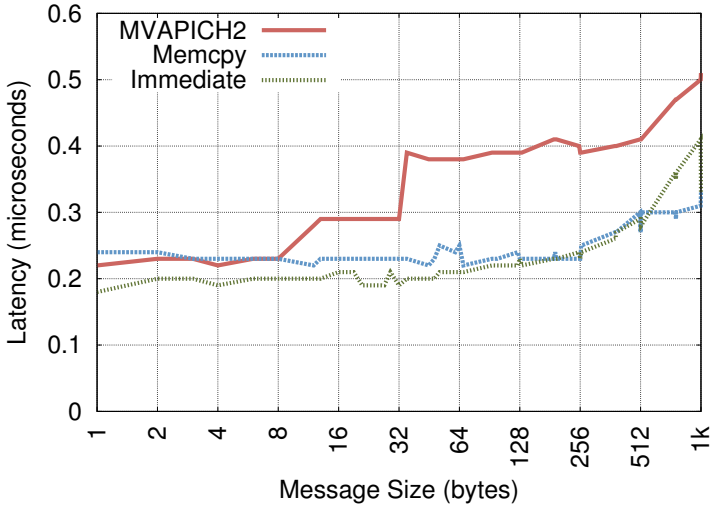
For small messages, the time saved by avoiding the cache miss more than makes up for the cost of the extra copy operation.

Figure 3.6 shows intra- and inter-socket small message latency on Sandy Bridge[1] using the NetPIPE **[86]** benchmark. Based on these results, we chose a message size threshold of 256 bytes, below which we use the immediate protocol. Above that, we use `memcpy()` or the synergistic protocol. Figure 3.7 shows small message latency on the Blue Gene/Q[1] system (one socket per node).

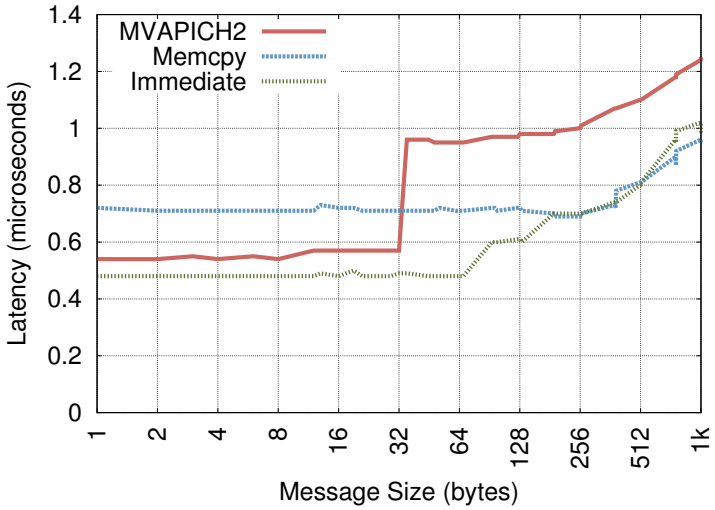
Since we observe no benefit from the immediate protocol on this system, we conditionally modify the sender-side protocol decision tree for Blue Gene/Q systems. Although no performance benefit is observed, it is still useful when the sender's buffer is not in shared memory (i.e., a stack or global variable). We can use the inline buffer and avoid allocating a temporary shared memory buffer, avoiding that overhead for small messages. Thus, on Blue Gene/Q, we first check if the sender buffer is not shared, and then if it is small enough, we use the immediate protocol.

3.2.4. Synergistic transfer protocol. For large messages, bandwidth is most important; constant-time matching latencies become less important as message size increases.

[1]Details on the Sandy Bridge and Blue Gene/Q systems can be found in Section 3.4.



(a) Intra-socket



(b) Inter-socket

FIGURE 3.6. Small message latency on Sandy Bridge.

We can achieve higher data transfer rates than possible with a single `memcpy()` by having both the sender and receiver participate in copying data from the send buffer to the receive buffer (Figure 3.8). To do this, we break the data into blocks and utilize a shared counter that is atomically updated by the sender and receiver. When the receiver matches a message, it initializes the counter (used as a byte offset) and begins copying data one block at a time. Before copying each block, the counter is incremented. If the sender enters the MPI

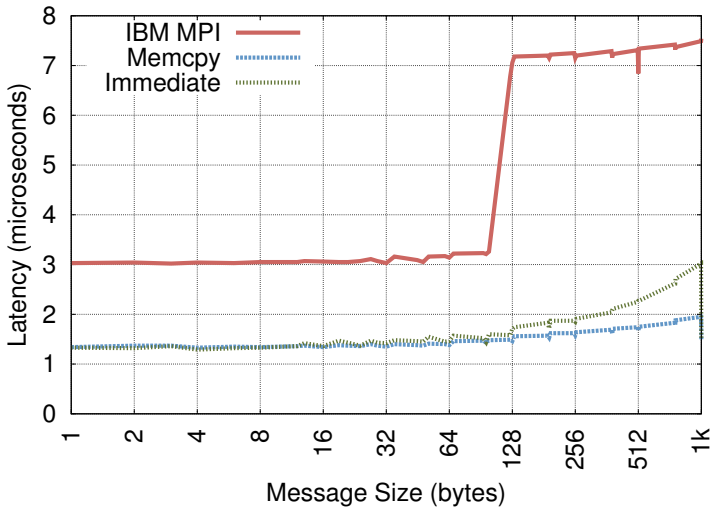


FIGURE 3.7. Small message latency on Blue Gene/Q.

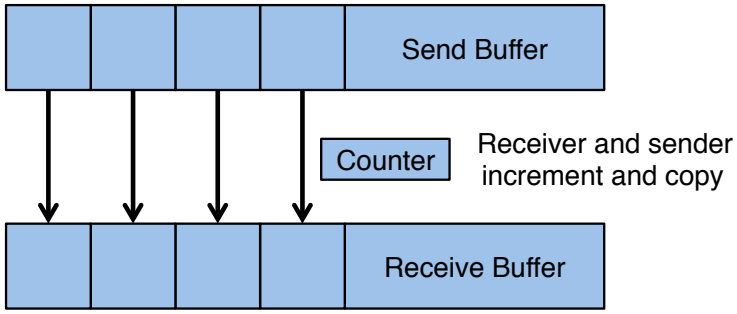


FIGURE 3.8. The synergistic protocol. After matching, the receiver begins copying the message one block at a time, incrementing a shared counter indicating the next block to copy. If the sender tests completion of its send and discovers the receiver is copying, it assists in copying blocks.

library and sees that the receiver is copying in block mode, it also begins incrementing the counter and copying blocks of data until the entire message has been copied.

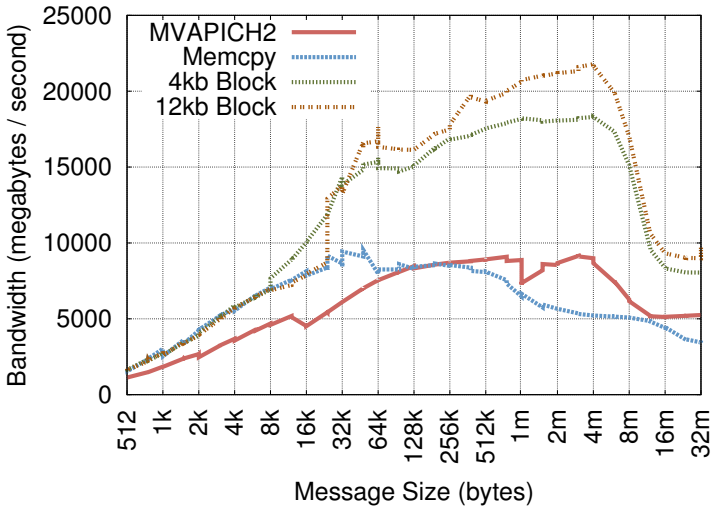
In the worst case, the sender does not participate (it is either executing application code or helping with other transfers), and we see the same bandwidth as a `memcpy()`, which is the peak bandwidth achievable by one core. The sender can enter and assist the transfer at any point. Bandwidth improvement then depends on when the sender begins assisting and on the peak bandwidth achievable by two cores.

The advantage of this protocol is that communication-computation overlap is greater than that of existing protocols when the sender has other work to do. Unlike the two-copy protocols used in current MPI implementations, the receiver can perform the entire data transfer without the sender, and does so when beneficial. Both the synergistic and immediate protocols allow looser synchronization between the sender and receiver compared to the pipelined two-copy MPI protocol; a full 'rendezvous' synchronization is not necessary. Communication performance is dynamically accelerated when the sender is able to assist the receiver in copying data.

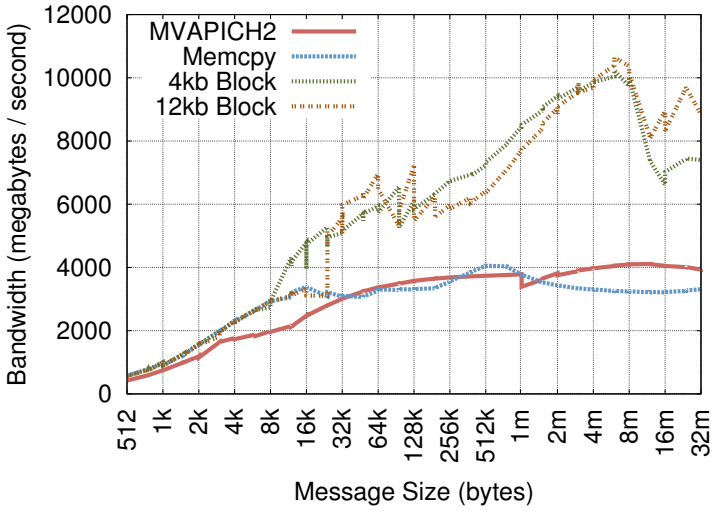
Alternate configurations of the synergistic protocol are possible. For example, we could perform matching on the sender, reversing the roles of the sender and receiver within the protocol. In that case, the receiver would optionally assist the sender with the data transfer. In our design, at any point a match is made, the receiver will copy the entire message before moving on to other work. Instead, we could do this only for blocking completion routines like `MPI_Recv()` and `MPI_Wait()`. Routines like `MPI_Irecv()` and `MPI_Test()` would attempt to match messages, but would not copy the data. With this approach, the receiver allows the sender to jump in and copy at any time after matching, but will not copy data itself until it must block waiting for completion of the receive operation.

Figure 3.9 shows intra- and inter-socket large message bandwidth on the Sandy Bridge system. Based on experimentation, we chose two different block sizes: 4kB and 12kB. For messages smaller than twice the block size, we use a single `memcpy()`, since the synergistic protocol needs multiple blocks to provide a benefit. Starting at 8kB, we use the synergistic protocol with a 4kB block. For messages greater than 24kB, we switch to a 12kB block. In some cases peak bandwidth is more than double that of MPI or `memcpy()`.

Figure 3.10 shows large message bandwidth on Blue Gene/Q, which has one socket per node. Based on experimental results, we chose block sizes of 16kB and 64kB.



(a) Intra-socket



(b) Inter-socket

FIGURE 3.9. Large message bandwidth on Sandy Bridge.

NetPIPE represents the ideal case for the synergistic protocol—both the sender and receiver are always ready and available to assist in data transfer. In practice, the bandwidth seen by applications will vary somewhere between that of `memcpy()` and the peak synergistic bandwidth depending on communication-computation overlap.

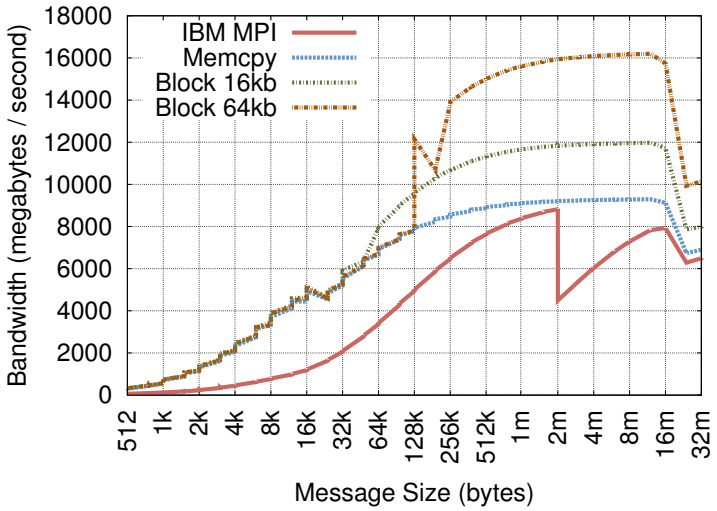


FIGURE 3.10. Large message bandwidth on Blue Gene/Q.

3.3. Communication Analysis

While raw communication performance is important, another way that MPI affects application performance is its effect on the cache and the application data structures within it [73]. We studied the effect that HMPI's and MPI's communication protocols have on the cache using the micro-benchmark in Figure 3.11, which models the typical interaction between the application and the MPI library. It reads the elements of a data buffer to bring it into the cache, then performs a ping-pong communication and finally reads the data buffer again. We conducted experiments on Sandy Bridge with buffer sizes between 128 bytes and 32kB (the size of Sandy Bridge's L1 Data cache) where the read loop either accesses buffer entries in `sequential` or `random` order and each cache line is accessed exactly once. Further, we studied configurations where `separate` buffers were used for both data and communication or a `common` buffer for both (in this case message size was $\leq$ buffer size). To understand how the different types of communication protocols affect the application's use of the cache we measure the number of cache misses the benchmark incurs during the second read loop.

Figure 3.12 compares the fraction of L1 data cache (32kB) misses on the Sandy Bridge system between HMPI and MVAPICH2. For each, we compute the fraction of data buffer

```
unsigned char *data_buf, *comm_buf;

// Read the buffer, bringing it into the cache.
for(int i = 0; i < x; i++)
    sum += data_buf[index(i)];

// Perform ping pong communication on either
// the data buffer (Common configuration) or a
// communication buffer (Separate)
if(Common)          Do_PingPong(data_buf);
else if(Separate) Do_PingPong(comm_buf);

// Read the buffer in sequential or random order,
// while measuring cache misses.
Start_Counters();
int sum;
for(int i = 0; i < x; i++)
    sum += data_buf[index(i)];
Measure_Elapsed_Counters();
```

FIGURE 3.11. Benchmark that models the impact of MPI communication on application cache use.

reads that miss: (number of L1 data misses) $*$ (cache line size)/(data buffer size). Then we compute the ratio $HMPI/MPI$, indicating the relative fraction of misses when reading the data buffer. Plots for four benchmark configurations are shown: `random` or `sequential` loop orders combined with whether the communication buffer is `common` or `separate` from the application buffer.

Values closer to 0 indicate HMPI has fewer misses, and are shown in lighter shades. Values close to 1.0 indicate HMPI and MPI have the same number of misses, shown in darker colors. For `random` reads, we see that HMPI induces fewer application misses across all data buffer sizes when the communication buffer is smaller than 8kB (one quarter of the L1 cache). The same is true for `sequential` reads for data buffers smaller than 16kB (one half of the L1 cache). For applications that operate on and communicate with buffers of a few kilobytes (expected to be the norm as the same amount of data is divided among more processor cores), HMPI has a significantly smaller impact on the application's use of the cache. The implication here is that using HMPI can result in better performance for

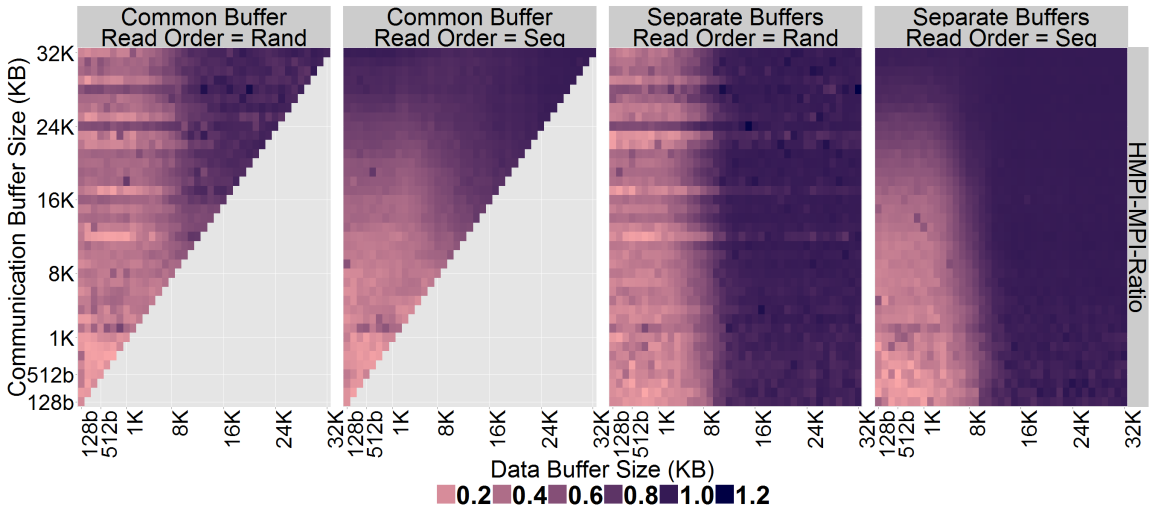


FIGURE 3.12. The fraction of accesses to the data buffer on which an L1 cache miss occurs with HMPI, divided by the same with MPI. Lighter colors indicate fewer misses in HMPI compared to MPI.

the non-communication parts of the application by not evicting application data from the cache.

3.4. Application Analysis

The various micro-benchmark results shown in Sections 2.1 and 3.3 give a picture of HMPI's shared memory communication performance in isolated scenarios. In this section, we compare the performance of HMPI to MPI for two applications: MiniMD and LULESH. We show results for one node (where our shared memory protocols are used exclusively in HMPI) as well as up to 64 nodes.

All figures in this section show 'percent improvement' on the y-axis calculated as $Y = 100 * (HMPI/MPI)$ using the respective HMPI and MPI wall clock times. We report the improvement in application time as well as the time taken specifically by each application's communication phases. For each individual experiment, we ran the code 20 times and reported results from the single run with the lowest application time. Unless indicated otherwise, all experimental results shown in this dissertation followed the same procedure.

The ratio of speedup between application and communication time varies based on the ratio of communication to computation in the application, which in turn depends on

several factors such as problem size and ratio of processing speed to memory bandwidth. All of our results show weak scaling with a fixed problem size per rank.

We show results for two different systems. Sandy Bridge ('Cab' at LLNL) has two Xeon ES-2670 [**18**] (eight core, 2.6 GHz) processors (16 cores total), 32 GiB of RAM per node, and a PSM Quad data rate (QDR) InfiniBand network. MVAPICH2 v1.9 was used as our comparison MPI on Sandy Bridge. The Blue Gene/Q system ('Sequoia' and 'Vulcan' at LLNL) has one PowerPC A2 [**38**] (sixteen core, four threads per core, 1.6 GHz) processor (64 tasks total) and 16 GiB of RAM per node. IBM's MPICH-based MPI library was used on Blue Gene/Q.

All HMPI results shown use our dlmalloc-based [**52**] 'shared heap' library, while the MPI results use the Linux default system allocator (which is also based on dlmalloc).

3.4.1. MiniMD. MiniMD is part of the Mantevo [**31**, **49**] mini-application suite, which consists of several mini-applications representing larger application classes. In particular, MiniMD is a simplified form of the more complex LAMMPS Molecular Dynamics (MD) code [**48**, **75**]. Such mini-applications are increasingly used in exascale research for their combination of simplicity and relevance. MiniMD is a MD simulation of a Lennard-Jones system, computing atom movement over a 3D space decomposed into a processor grid. The primary work loop performs the following steps every iteration:

(1) Exchange position information of atoms in boundary regions with up to six neighboring ranks.
(2) Compute forces from both local atoms and those in boundary regions from neighboring ranks.
(3) Exchange force information of atoms in boundary regions to neighboring ranks.
(4) Update local atom velocities and positions.

Appendix A contains code listing for the communication phases (steps 2 and 4) We ran 2,500 iterations and used weak scaling with approximately 1,000 atoms. The MiniMD problem size is determined by the formula $||\sqrt[3]{NP * NA/4}||$, where NP is the number of ranks and NA is the number of atoms per rank. Since the problem size must be rounded

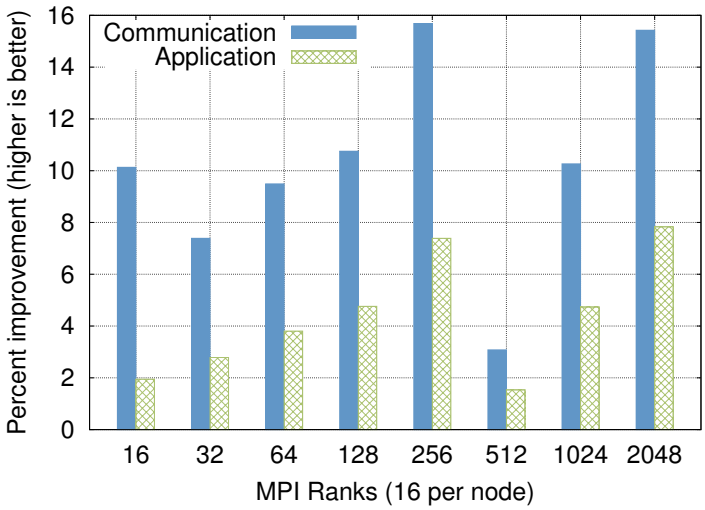
Ranks	Total	% Local	Immediate	Memcpy	Synergistic	% Assist
16	508,320	100.00%	22,386	325,278	160,656	92.43%
32	1,024,704	67.19%	35,680	12,704	640,128	84.99%
64	2,065,536	66.67%	81,710	15,058	1,280,256	84.61%
128	4,131,072	50.00%	107,241	661,463	1,296,832	98.86%
256	8,262,144	50.00%	220,536	69,768	3,840,768	98.46%
512	16,524,288	50.00%	492,739	87,869	7,681,536	98.58%
1,024	33,048,576	33.33%	564,760	209,384	10,242,048	93.57%
2,048	66,097,152	33.33%	1,142,216	406,072	20,484,096	94.88%

FIGURE 3.13. MiniMD message counts for HMPI protocols on Sandy Bridge. Total is the count of all messages, local and remote. % Assist is the portion of synergistic transfers assisted by the sender.

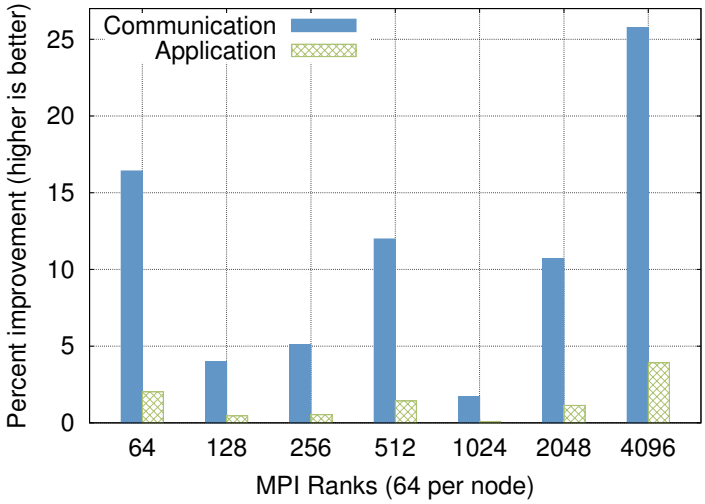
to an integer, the actual number of atoms per rank can vary slightly depending on the number of ranks.

Table 3.13 shows statistics on how many messages are passed during a MiniMD run and via which protocols on Sandy Bridge. As discussed previously (Section 3.2), the protocol is chosen based on message size. The Memcpy column counts single-copy transfers of messages too large for the immediate protocol, but too small for the synergistic protocol. We see that all three protocols are used to varying degree as the number of ranks increases. A consistently high percentage of the synergistic protocol transfers are assisted by the sender, indicating that our synergistic protocol is providing valuable speedups over `memcpy()` for MiniMD. Only 1.1-15.4% of synergistic protocol transfers are not assisted by the sender.

Figure 3.14 shows performance comparisons for MiniMD. On Sandy Bridge, we observe communication speedups ranging from 5.0-18.5%, resulting in total application time improvements of 0.9-9.6%. Blue Gene/Q shows improvements of 1.7-25.8% communication time and 0.1-3.9% application time. Compared to Sandy Bridge, Blue Gene/Q has more memory and network bandwidth per FLOP. As a result, a smaller portion of execution time is spent communicating on Blue Gene/Q. Thus, changes in communication time have a smaller impact on overall application performance. On both systems we see a wide range in speedups depending on the number of ranks. Not only are the number



(a) Sandy Bridge (16 ranks/node)



(b) Blue Gene/Q (64 ranks/node)

FIGURE 3.14. HMPI performance gains relative to MPI for MiniMD.

of ranks changing, the problem size per rank changes slightly, and more importantly, message sizes. It appears that MiniMD decomposes the 3D space differently depending on the number of ranks, splitting across more axes as the number of ranks increases. The change in decomposition is reflected in the local communication column of Figure 3.13 where the ratio of locality makes step-wise changes, as well as the relative performance gains between varying numbers of ranks in Figure 3.14.

Ranks	Total	% Local	Immediate	Memcpy	Synergistic	% Assist
8	21,656	100.00%	2,432	16,824	2,400	34.38%
64	339,336	41.12%	144	120,192	19,200	50.79%
512	3,588,536	27.17%	896	833,408	140,800	60.32%
1,728	13,211,176	18.35%	1,584	2,034,288	388,800	48.49%

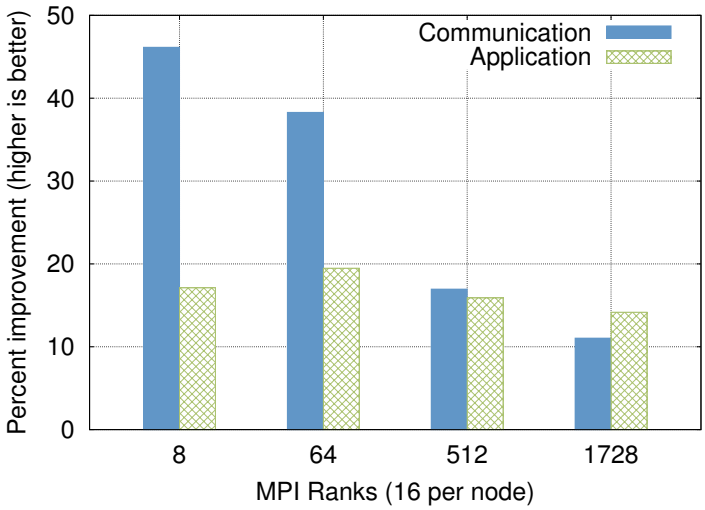
FIGURE 3.15. LULESH message counts for HMPI protocols on Sandy Bridge. Total is the count of all messages, local and remote. % Assist is the portion of synergistic transfers assisted by the sender.

3.4.2. LULESH. LULESH, also known as Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics [**42**,**51**], is the mini-application version of a full-size hydrodynamics code (ALE3D [**50**]) in use at Lawrence Livermore National Laboratory (LLNL). LULESH simulates the Sedov blast wave problem on a uniform 3D mesh decomposed spatially among MPI ranks. In each time step, multiple exchanges with up to 27 neighbors are performed. Each data exchange is implemented using non-blocking sends and receives.

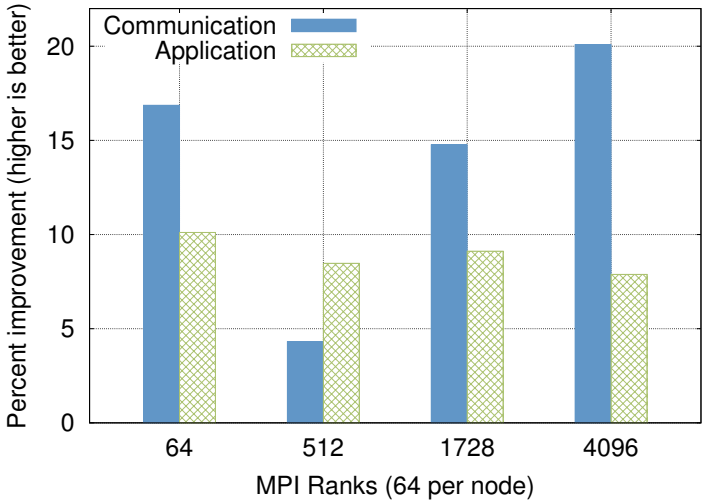
Table 3.15 shows statistics on how many messages are passed during a LULESH run and via which protocols on Sandy Bridge. Although both MiniMD and LULESH are MD codes, their communication characteristics are clearly very different. Here, the percentage of local communication decreases more significantly as the number of ranks increases. The majority of messages fall between the thresholds for the immediate and synergistic protocols, so `memcpy()` is the most common communication mechanism. When messages are large enough to use the synergistic protocol, the sender accelerates the transfer 34-48% of the time.

Figure 3.16 shows performance comparisons for LULESH. LULESH requires that the number of ranks be a perfect cube (i.e., $NP = x^3$). A fixed problem size of 15^3 per rank was used. On Sandy Bridge, we see communication time speedups of 11-46.1% and application time speedups of 14.1-19.5%. Communication time speedups of 4.2-20.1% and application time speedups of 7.9-10.1% are seen on Blue Gene/Q. As we saw with MiniMD, communication time speedups have a smaller impact on Blue Gene/Q compared to Sandy Bridge.

3.4.3. Application Cache Locality. In addition to wall clock performance, we also compared L2 and L3 cache misses incurred by the entire application when using HMPI



(a) Sandy Bridge (16 ranks/node)

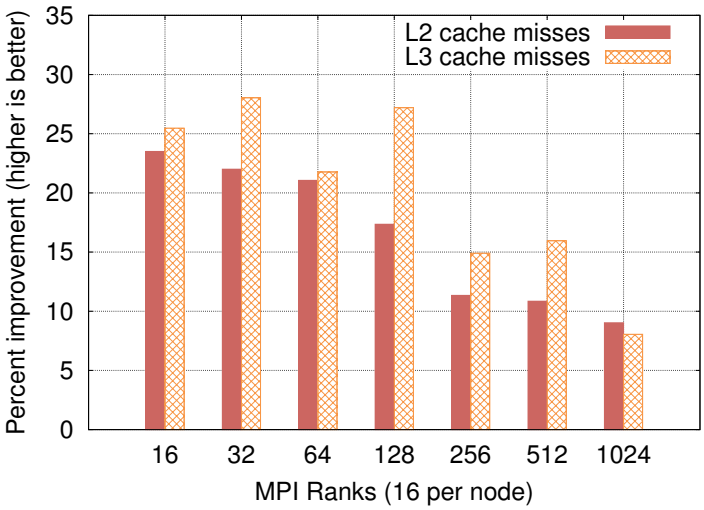


(b) Blue Gene/Q (64 ranks/node)

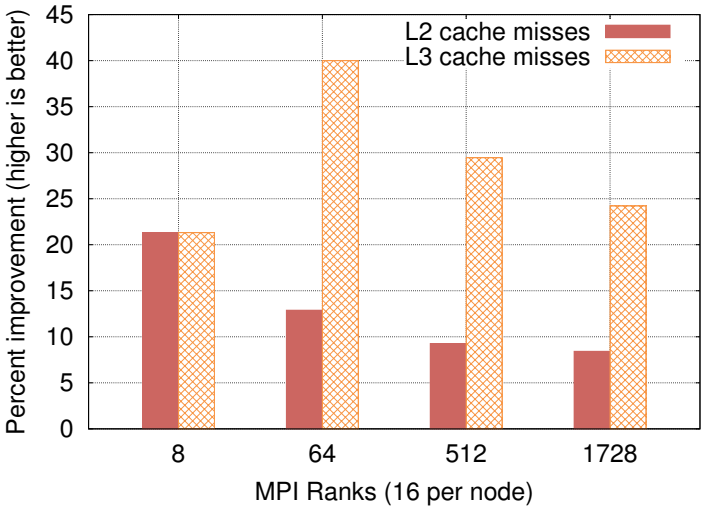
FIGURE 3.16. HMPI performance gains relative to MPI for LULESH.

and MVAPICH2. Figure 3.17 shows results on the Sandy Bridge system for MiniMD and LULESH, respectively. Sandy Bridge's processors have three levels of cache. L1 (32kB) and L2 (256kB) are exclusive to each core, while L3 (20mB) is shared by all 8 cores.

HMPI reduces both L2 and L3 cache misses for both applications. We observe reductions of 8-28% for MiniMD and 8.3-40% for LULESH. In LULESH, the 8 rank L3 cache results appear to be an outlier. Sandy Bridge has 16 cores per node, so only half of the



(a) MiniMD



(b) LULESH

FIGURE 3.17. Reduction of total application cache misses when using HMPI compared to MPI on Sandy Bridge.

available cores are utilized. More L3 cache is available per rank, resulting in fewer total misses for both HMPI and MPI.

3.5. Conclusion

In this chapter, we showed multiple ways to improve on the state of the art in intra-node MPI communication protocols. The existing pipelined two-copy protocol forces shared memory into the narrow abstraction of a network interface. Recent work on the LiMIC and KNEM kernel extensions provide a mechanism for single-copy data transfers within MPI , but still have their drawbacks. Namely, they are not found by default on Linux systems and they require making system calls into the kernel, limiting generality.

We utilize HMPI and a shared heap to perform single-copy data transfers without requiring kernel extensions, and avoided the overhead associated with kernel calls. Thus, we improve on prior work by improving portability and making single-copy feasible for small message sizes. Going further, we developed the *immediate* and *synergistic* protocols to achieve performance improvements over single-copy. For small messages, we found that a two-copy scheme reduces communication latency by improving data locality to reduce cache misses. Bandwidth can be increased when transferring larger messages by utilizing two cores (the sender *and* receiver) to copy data.

Experimentally, we demonstrate communication time speedups of up to 26.2% and 46.1% for the MiniMD and LULESH mini-applications, respectively. In addition to communication performance, we showed how our communication design in HMPI have a lesser impact on the processor cache, allowing more of the application's data structures to remain resident in the cache hierarchy. All of the techniques shown in this chapter transparently improve performance for existing MPI applications on modern and future multi-core High Performance Computing (HPC) systems.

4

Ownership Passing

In prior chapters, we have shown techniques for better utilizing shared memory to improve intra-node message passing performance. Any MPI application can transparently benefit from that work, without modification. Such research has been the traditional method for improving parallel application scaling and performance, but is restricted to the confines of the MPI standard, which was designed for distributed memory computers with either single-core or small SMP nodes.

On today's architectures, the current copy-based message-passing model is suboptimal in terms of *memory* (send and receive buffers), *energy* (data movement consumes most energy [**4**]), and *time* (busses are used twice which reduces performance). To avoid those

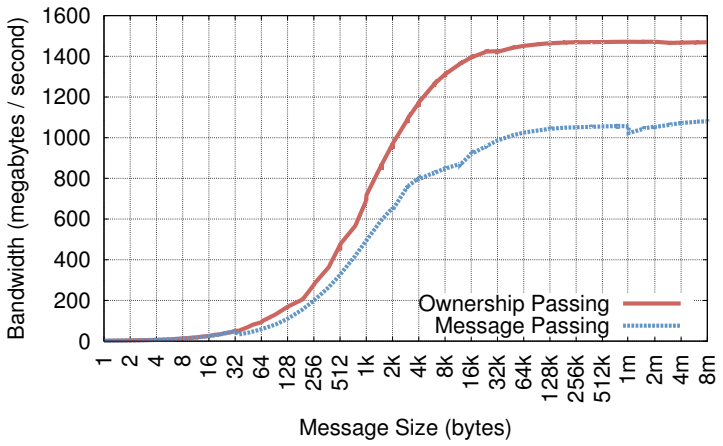


FIGURE 4.1. Bandwidth of Ownership Passing vs. Message Passing.

issues, many software developers switched to hybrid programming techniques, combining MPI for inter-node communication with shared memory programming models such as OpenMP [**80**] for intra-node communication. However, achieving the same level of performance is a tedious and complex task and often requires major code restructuring to work in the shared memory world [**81**].

In this chapter, we use an ownership passing technique to easily and safely transform message-passing parallel applications to utilize shared memory hardware more efficiently. Instead of re-writing existing applications, we simply change them to pass a pointer from the sender to the receiver instead of copying the data. In fact, the production and consumption of data buffers in our system automatically aligns in a pipelined fashion so that both stages can overlap.

As a motivating example, we show the effective bandwidth when passing a memory buffer instead of copying its contents on the Sandy Bridge system in Figure 4.1. The measurement was done with the well-known NetPIPE [**86**] ping-pong benchmark on the Sandy Bridge system (described in Section 3.4). To ensure fair comparison we extended NetPIPE to read the received data, thus accounting for the cost of transferring the data using ownership passing. We see that the standard copy approach is limited by the memory copy bandwidth and synchronization costs while ownership passing essentially requires only

synchronization and reading from (potentially remote) memory. Thus, ownership passing can be significantly faster than standard message passing on today's multi-core systems.

Our approach is true *zero-copy* (*zero-touch*, in fact) because the buffer contents are neither read nor written during the communication. We develop a novel memory pooling technique to re-use communication buffers and avoid synchronization.

The contributions of this chapter are:

- An interface for ownership passing that is compatible with MPI and allows for easy porting from MPI codes.
- Analysis of the performance of ownership passing in realistic environments.
- Demonstration of practical results with important HPC micro-applications and application kernels that have been improved with our technique.

In the next section, we describe the ownership passing technique. Section 4.2 introduces our Ownership Passing Interface (OPI) design and API. We describe and give examples for applying ownership passing to point-to-point message passing codes in Section 4.4 and collective communication in Section 4.5. Finally, Section 4.7 provides a performance analysis of ownership passing using a micro-benchmark and several applications.

4.1. Ownership Passing Semantics

MPI's distributed memory design ensures that only one rank can access a buffer (any arbitrary memory region). That is, exactly one rank *owns* a buffer, and that is the only rank that may read or write that buffer. A shared heap, as discussed in Chapter 2, makes it possible for ownership to be *passed* from one rank to another via message passing. Ownership passing works because the same memory is accessible on all ranks within a node. When a rank gives away ownership of a buffer, that rank can no longer access that buffer. Likewise, taking ownership of a buffer enables exclusive read and write access.

Ownership passing reinterprets the concept of distributed memory in a way that retains its simplicity while taking advantage of shared memory hardware. Traditional distributed memory partitions the application's address space into static private memory blocks. In contrast, ownership passing allows this partition to be dynamic, with memory regions

moving from one private space to another while maintaining the invariant that each memory region is privately owned by exactly one thread of execution. The resulting flexibility makes it possible for message passing applications to utilize shared memory hardware in ways similar to native shared memory applications but without concern about data races and other complications of the shared memory model. Not only is our approach beneficial on cache-coherent architectures (e.g., commodity x86), it also works on non-coherent cache and Non-Uniform Memory Architectures (NUMAs).

4.2. Ownership Passing Interface (OPI)

Transferring ownership of a buffer only requires sending a pointer instead of the entire message data. When the new owner of a buffer begins reading the message from its original location, the shared memory hardware will begin to stream data from the sender rank's cache and/or memory to the receiver. Standard architectural processor features such as cache snooping and memory prefetching improve the performance and enable efficient communication/computation overlap in hardware. Since the communication library never accesses the buffer data, true *zero-copy* communication is achieved.

We define a small extension Application Programming Interface (API), referred to as the Ownership Passing Interface (OPI), to simplify the use of ownership passing in applications. OPI, as described here, has been implemented in HMPI. The C interface is shown in Figure 4.2 and Figure 4.3 shows a simple MPI example and its OPI counterpart. All routines are thread-safe with respect to one another.

`OPI_Alloc()` and `OPI_Free()` allocate and deallocate new communication buffers, calling `malloc()` and `free()` and performing additional management of buffer memory pools, as described in Section 4.3. Ownership passing is performed using the `OPI_Give()` and `OPI_Take()` routines, which are analogous to `MPI_Send()` and `MPI_Recv()` (non-blocking versions, `OPI_Give()` and `OPI_Take()`, are also available similarly to non-blocking MPI calls).

`OPI_Give()` consumes the provided buffer. If the destination rank is in the same address space (on the same node), then the source rank synchronizes with the destination

Function Prototype	Behavior
`int OPI_Alloc(void** ptr, size_t length)`	Allocate a new communication buffer of some length.
`int OPI_Free(void** ptr)`	Release a buffer allocated by `OPI_Alloc()`.
`int OPI_Give(void** ptr, int count, MPI_Datatype datatype, int rank, int tag, MPI_Comm comm)`	Pass ownership of a buffer to another MPI rank (blocking form).
`int OPI_Igive(void** ptr, int count, MPI_Datatype datatype, int rank, int tag, MPI_Comm comm, MPI_Request req)`	Pass ownership of a buffer to another MPI rank (non-blocking form).
`int OPI_Take(void** ptr, int count, MPI_Datatype datatype, int rank, int tag, MPI_Comm comm, MPI_status st)`	Receive ownership of a buffer from another MPI rank (blocking form).
`int OPI_Itake(void** ptr, int count, MPI_Datatype datatype, int rank, int tag, MPI_Comm comm, MPI_Request req)`	Receive ownership of a buffer from another MPI rank (non-blocking form).

FIGURE 4.2. Non-blocking Ownership Passing Interface (OPI).

MPI

```
double* buf = malloc(size);

if(rank == 0) {
    pack(buf, size, ...);
    MPI_Send(buf, 1, ...);
} else if(rank == 1) {
    MPI_Recv(buf, 0, ...);
    unpack(buf, size, ...);
}

free(buf);
```

Ownership Passing

```
double* buf;

if(rank == 0) {
    buf = OPI_Alloc(&buf, size);
    pack(buf, size, ...);
    OPI_Give(&buf, 1, ...);
} else if(rank == 1) {
    OPI_Take(&buf, 0, ...);
    unpack(buf, size, ...);
    OPI_Free(&buf);
}
```

FIGURE 4.3. Example of MPI to OPI conversion.

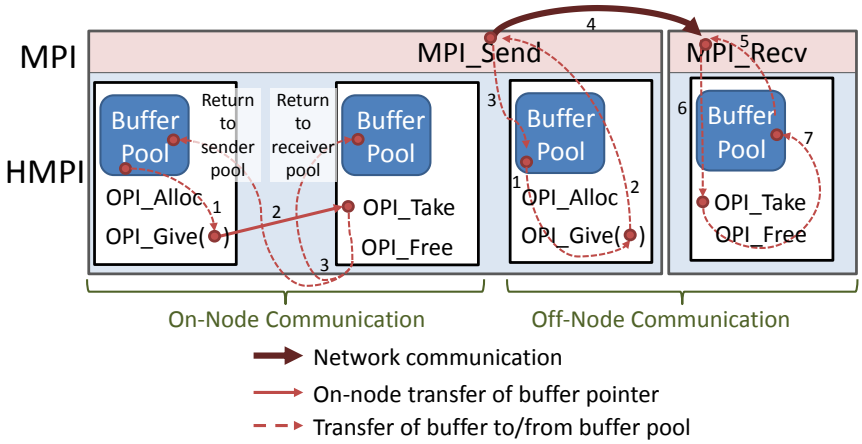


FIGURE 4.4. Flow of control and buffers in ownership passing.

and passes the buffer. Otherwise, the source rank invokes a normal `MPI_Send()` call with the given arguments and calls `OPI_Free()` using the buffer after the send completes.

`OPI_Take()` produces a new buffer for the receiver. If the buffer comes from a rank in the same address space, then it will simply return the pointer to the buffer. If the source is a rank from a different address space then it allocates memory of the required size, invokes `MPI_Recv()` on this buffer, and returns the buffer upon completion of the remote receive.

Note that `OPI_Alloc()` and `OPI_Take()` introduce new buffers, while `OPI_Give()` and `OPI_Free()` relinquish ownership of a buffer. For safety and to promote the *ownership* concept, the latter two routines clear the provided pointer to `NULL` before returning. In the case of `OPI_Give()`, we could instead return the original buffer if the destination rank is not on the same node, allowing the application to potentially reuse the buffer. We chose not to do so because this breaks our defined semantics for the give operation, plus utilizing such a feature would add complexity for little benefit.

Figure 4.4 depicts the ownership passing mechanism, flow control, and buffers. Within a (left side), buffers are produced by `OPI_Alloc()`, flow through `OPI_Give()` and `OPI_Take()`, and finally consumed by `OPI_Free()`. See the next section for discussion of the buffer pools. Between nodes, `OPI_Give()` emulates ownership passing using `MPI_Send()` and `OPI_Free()` while `OPI_Take()` uses `MPI_Recv()` and `OPI_Free()`.

In our HMPI implementation, we support matching of OPI give and take operations with MPI send and receive operations. For example, a sender could use `OPI_Give()` to give away a buffer, while the receiver might receive that message date using `MPI_Recv()`. That is, ownership passing messages are not matched separately from existing MPI (See Section 3.1 for discussion on MPI matching rules). We support this functionality by falling back to copying when OPI operations are not used for both sides of the message transfer. `OPI_Alloc()` and `OPI_Free()` are used appropriately for buffer management, just as in the off-node ownership passing case.

4.3. Communication Buffer Management

Once a rank has taken ownership of a buffer and consumed its contents, that buffer must be disposed of. We could simply free the buffer back to the heap, but this is not ideal. `Malloc()` and `free()` must be implemented in a thread-safe manner, since ownership passing makes it likely that one rank will free memory allocated by some other rank. For our shared heap allocator, this means a lock is needed to protect each process's heap state. Ownership passing encourages a producer-consumer pattern where a new buffer is allocate for every message to be sent. As we will see, `malloc()` and `free()` are costly library calls, so we must minimize calling them to optimize performance.

We can alleviate the costs of `malloc()` and `free()` by caching buffers in a memory pool. When allocating, we search a memory pool for the first buffer large enough for the requested size and reuse it.[1] If no such buffer exists, a new one is allocated. When freeing a buffer, we return it to a memory pool instead of the heap. In practice, `malloc()` calls will occur frequently at the beginning of the application's execution. As the application settles into a working set of buffers, buffers are efficiently reused.

Using one memory pool per node would require a lock shared between all ranks on a node, which is not an improvement over using the heap. Instead, we maintain one memory pool per rank. Now, a choice must be made—buffers can be returned to either the sender's or the receiver's memory pool. Returning buffers to the sender's pool requires

[1] All sizes are rounded up to the next multiple of the OS page size to improve memory reuse.

a lock, since multiple ranks may simultaneously return a buffer to the pool, perhaps also while the sender is allocating. However, this approach distributes contention over many locks rather than one, yielding an improvement.

On the other hand, no lock is required if we return buffers to the receiver's local pool—each rank only accesses its own memory pool. Internally, a memory pool is implemented as a linked list. The list is maintained in Most Recently Used (MRU) order: when a buffer is freed, it is inserted at the head of the list. When allocating, the list is searched starting from the head for a large enough buffer. This design has good cache locality due to the fact that it prefers using buffers that are more likely to be present in the cache.

Two issues can occur when using a memory pool scheme. If one rank sends (in the case of a sender pool) or receives (receiver pool) more messages than others (i.e., an asymmetric communication pattern), buffers accumulate in one memory pool and never get reused, wasting memory. Similarly, if an application begins requesting larger buffer sizes, the smaller buffers will sit unused in the memory pool and are never freed. We solve both of these issues with one solution. If the number of buffers in a pool exceeds some threshold (checked each time a buffer is returned to the pool), some buffers are freed back to the heap. This memory may eventually be reused in later `malloc()` calls.

To evaluate the performance of these different buffer management schemes, we measured the sum of the time taken by `OPI_Free()` and `OPI_Alloc()` routines in our ownership passing microbenchmark (see Section 4.7.1). For this experiment we allocated 8 byte buffers and averaged the time over 5,000 benchmark iterations. The results, shown in Figure 4.5, demonstrate that returning buffers to the receiver's pool has, by far, the lowest cost. For Sandy Bridge, we show timings where the two ranks are on the same socket and on different sockets. The cross socket L3 cache misses indicate accesses of memory owned by a remote rank: remote heap data structures or the sender's memory pool. Blue Gene/Q has only once socket and only two levels in its cache hierarchy.

Malloc/free is more expensive for several reasons: (i) it uses a more memory management algorithm, (ii) may use the `sbrk()` and `mmap()` system calls, and (iii) requires locking to protect data structures (so that one rank can free memory allocated by another

	Scheme	Time (ns)	Cache Misses L1	L2	L3
Sandy Bridge Same Socket	Malloc/Free	306	36.09	17.37	0.00
	Sender Pool	198	12.24	5.33	0.00
	Receiver Pool	12	7.30	1.02	0.00
Sandy Bridge Cross Socket	Malloc/Free	619	28.89	7.50	7.42
	Sender Pool	570	11.65	5.03	5.02
	Receiver Pool	12	7.30	1.02	0.00
Blue Gene/Q Single Socket	Malloc/Free	1006	35.27	0.00	N/A
	Sender Pool	368	27.55	0.00	N/A
	Receiver Pool	185	27.50	0.00	N/A

FIGURE 4.5. Average time (nanoseconds) and cache miss counts of several buffer management schemes when allocating and freeing an 8 byte buffer via OPI. Lower values are better.

rank). The receiver pool scheme exploits data locality and minimizes the frequency of expensive `malloc()` and `free()` calls. We use this scheme for all experimental results shown later in the paper.

4.4. Point to Point Ownership Passing

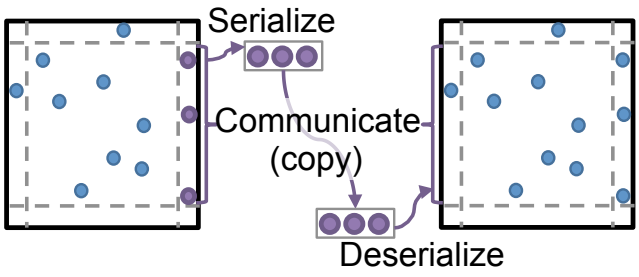
Transforming an MPI application to use ownership passing consists of three steps:

(1) Replace `MPI_Send()` and `MPI_Recv()` and related communication functions with `OPI_Give()` and `OPI_Take()`, respectively. OPI makes this trivial; except for the additional referencing in the first argument (buffer pointers) `OPI_Give()` and `OPI_Take()` accept the same arguments as `MPI_Send()` and `MPI_Recv()`.

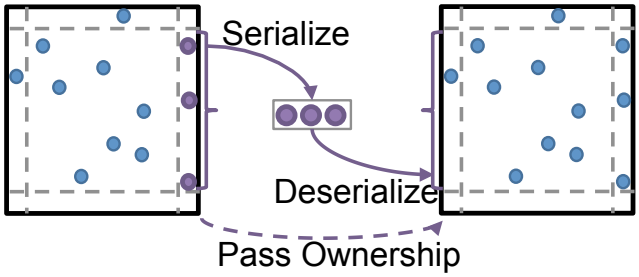
(2) Insert a call to `OPI_Alloc()` before packing a communication buffer for sending. Since giving away ownership consumes the communication buffer, a new one must be allocated every time a message is sent.

(3) Insert a call to `OPI_Free()` after receiving and unpacking a communication buffer. Since taking ownership produces a new communication buffer, every received message must be freed.

Although communication buffers can be allocated at any time before they are packed and can be freed any time after they are unpacked, the best buffer reuse is achieved by allocating send buffers as 'late' as possible in the application (immediately before they're



(a) MPI



(b) Ownership Passing

FIGURE 4.6. Molecular dynamics overlap communication. The boundary particles must be serialized into contiguous buffers. Ownership passing eliminates data copying during communication.

packed), and freeing received buffers as 'early' as possible (immediately after they're unpacked).

To illustrate the changes needed to perform ownership passing with MPI, we present a two-dimensional molecular dynamics (MD) example, where space is divided into regions and each processor is responsible for computing forces on and positions of particles within its region. Particles on the boundary of each processor's region are communicated to processors responsible for adjacent space regions. Figure 4.6(a) shows how this is performed using MPI for a single boundary exchange (the same is done with other neighbors). Boundary particles are serialized into a buffer, which is then sent via MPI (copied) and deserialized on the receiver. Ownership passing, shown in Figure 4.6(b), speeds up the process by replacing the MPI copy with a transfer of ownership of the packed buffer.

Figure 4.7 presents pseudo-code of the communication process for MPI and ownership passing. The ownership passing version allocates a buffer from the local memory pool just

MPI

```
pack_particle_buffer(send_buffer, particle_data);

MPI_Isend(send_buffer, count, datatype,
    neighbor_rank, TAG, MPI_COMM_WORLD, &reqs[0]);

MPI_Irecv(&recv_buffer, count, datatype,
    neighbor_rank, TAG, MPI_COMM_WORLD, &reqs[1]);

MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);

unpack_particle_buffer(recv_buffer, particle_data);
```

Ownership Passing

```
OPI_Alloc(&send_buffer, max_particles);

pack_particle_buffer(send_buffer, particle_data);

//Pass ownership of our send buffer.
OPI_Igive(&send_buffer, count, datatype,
    neighbor_rank, TAG, MPI_COMM_WORLD, &reqs[0]);

//Take ownership of a new receive buffer.
OPI_Itake(&recv_buffer, count, datatype,
    neighbor_rank, TAG, MPI_COMM_WORLD, &reqs[1]);

MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);

unpack_particle_buffer(recv_buffer, particle_data);

//Always return the receive buffer.
OPI_Free(&recv_buffer);
```

FIGURE 4.7. Boundary exchange communication between a pair of neighbors.

before packing, regardless of whether the destination is local or remote. Ownership is passed if the neighbor rank is local; otherwise the message is sent as would normally be done via MPI with negligible overheads. After unpacking, the received buffer is returned to the receiver's memory pool.

In this dissertation, we only consider the manual transformation of codes to use our optimization techniques. The scope of our work is to develop performance optimizations for improving MPI shared memory performance and implement them in a runtime library

(HMPI). That said, we have collaborated to develop compiler techniques for automatically transforming MPI point-to-point communication to use ownership passing [**24**]. Work is ongoing to implement the automatic transformation techniques using the ROSE [**78**, **79**] source-to-source compiler tool.

4.4.1. MPI Datatypes. MPI datatypes allow strided sequences of elements or arbitrary memory layouts to be sent and received. The ownership passing principle can be adapted to communication of disjoint sets of elements. Since MPI datatypes must be specified at both the sender and the receiver, ownership to the elements specified by the datatype can be easily transferred as long as the datatypes on both sides have the same memory layout. In cases where the receiver only consumes part of a buffer or when the sender wishes to reuse the data after passing ownership, the receiver can pass ownership back. Such an approach is analogous to protecting access to the buffer with a mutex. MPI also allows applications to provide different datatypes on the sender and receiver that place data in memory in different orders (e.g., matrix row on the sender and matrix column on the receiver). Copying is required to support this use-case and is the most efficient way to provide this specialized functionality.

4.5. Collective Ownership Passing

Although we specifically focus on ownership passing for point-to-point communication, it can also be used effectively for collective communication. Here, we briefly discuss how ownership passing can be used to implement scatter, gather, and all-to-all.

First, consider the MPI scatter operation in which applications allocate and pack into one send buffer, with respective portions to be scattered to each rank. Ownership of this buffer can be passed to the receive ranks as a collective (using the existing `MPI_Scatter()`), but this approach raises the question of which rank should release the buffer, and when. Synchronization (e.g., a barrier) is required to solve this problem, but can negate the performance benefits of ownership passing.

A better solution is to allocate a separate buffer for each destination rank, as illustrated in Figure 4.8. `OPI_Igive()` is used to pass ownership of each buffer to its respective rank.

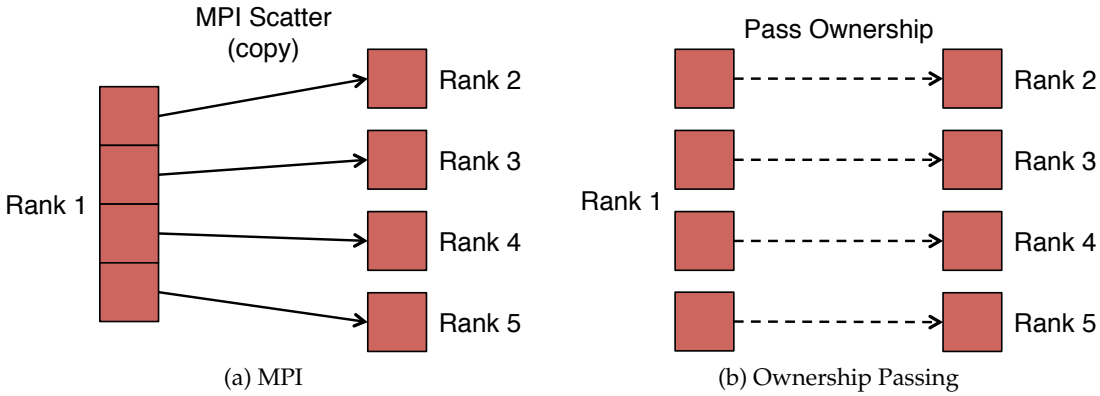


FIGURE 4.8. Scatter collective communication. MPI copies out of one buffer, while ownership passing gives separate buffers to each rank.

Each receiver can then return its buffer to the sender's memory pool without a global synchronization.

Figure 4.9 demonstrates ownership passing for scatter communication in code form. Note that we have used point-to-point communication, although ownership could also be transferred using an `MPI_Scatter()` routine or OPI equivalent. Gather operations are performed in a similar manner to scatter; the root rank gathers an array of buffer pointers to take ownership. When finished with the data, the root can release each buffer back to the respective memory pools. An ownership passing all-to-all can be constructed by combining scatter and gather: each rank gives a buffer to every rank, and simultaneously takes a buffer from each rank.

4.6. Further Techniques

Our work so far has been built around the single owner invariant: exactly one rank (process) has permission to read or write a memory region at a time. However, this is purely a semantic definition to maintain MPI's distributed memory abstraction and keep an intuitive design for the OPI extension. Although simple, this strict definition of ownership potentially limits further optimization possibilities in applications.

In reality, memory can be accessed by any rank at any time. Recent work utilizing ownership passing **[61]** has shown how reference-counted buffers can be used to enable

```
if(my_rank == root) {
  for(int i = 0; i < mpi_size; i++) {
    //Allocate a buffer and pack data for rank i
    OPI_Alloc(&buffer, buffer_size);

    pack_buffer(buffer, i);

    //Pass ownership of the buffer.
    OPI_Igive(&buffer, count, datatype,
        i, TAG, shared_mem_comm, &reqs[i]);
  }
}

OPI_Itake(&buffer, count, datatype,
    root, TAG, shared_mem_comm, &recv_req);

//Wait to take ownership from the root.
MPI_Wait(&recv_req, MPI_STATUS_IGNORE);

unpack_buffer(buffer);

//Always return the buffer to where it came from.
OPI_Free(&buffer);

//Complete the send requests.
MPI_Waitall(mpi_size, reqs, MPI_STATUSES_IGNORE);
```

FIGURE 4.9. Ownership passing in scatter collective communication.

multiple readers and 'cloning' data when necessary due to writes. The same distributed memory abstraction is maintained, using ownership passing and allowing access by ranks whenever possible, and copying data only when necessary.

Standard thread synchronization tools such as Portable Operating System Interface (POSIX)-threads mutexes, condition variables, and semaphores **[10]** work as-is across processes via the heap. Reader-writer locks **[36, 46, 55, 64]** can be used to give multiple ranks read-only access to memory, while still allowing exclusive write ownership. Finally, MPI itself can be used to provide synchronization through `MPI_Barrier()`, small point-to-point messages, etc.

Taking advantage of these possibilities can yield even further performance gains, but is outside of the scope of this dissertation. Our goal is to more effectively utilize shared memory in a distributed programming model, which our more restricted definition of *ownership* accomplishes. Further work along these lines means programming to both shared and distributed memory programming models in a similar fashion to hybrid MPI+OpenMP methods. Potential optimizations with a more relaxed memory model would be highly application-specific, our work focuses on more generally applicable message passing optimizations.

4.7. Experimental Results

Experimental results were obtained in the same environment as described previously in Section 3.4. Namely, we used the Sandy Bridge (16 ranks per node) and Blue Gene/Q (64 ranks per node) systems. OPI is implemented as part of our HMPI library, and all codes discussed in this chapter were transformed by hand. In the following section, we provide analysis of ownership passing with a microbenchmark. Section 4.7.2 revisits the MiniMD mini-application, while Section 4.7.3 shows results for a 2-dimensional FFT code.

4.7.1. Microbenchmark Analysis. We developed OPBench (Ownership Passing Benchmark) to analyze the performance characteristics of ownership passing (as implemented by the OPI interface) and compare them to MPI and HMPI. Without modifications, a standard microbenchmark like NetPIPE [**86**] doesn't measure time taken to access the associated buffers before and after communication. Normally this practice is fine, as it focuses solely on network performance. However, ownership passing has the unique feature that communication latency is constant regardless of message size. In fact, bandwidth (as would be reported by, say, NetPIPE) can be modeled using the formula $Bandwidth = MessageSize/Latency$, where $Latency$ is always the time to send an 8 byte (size of a pointer) message from one rank to another.

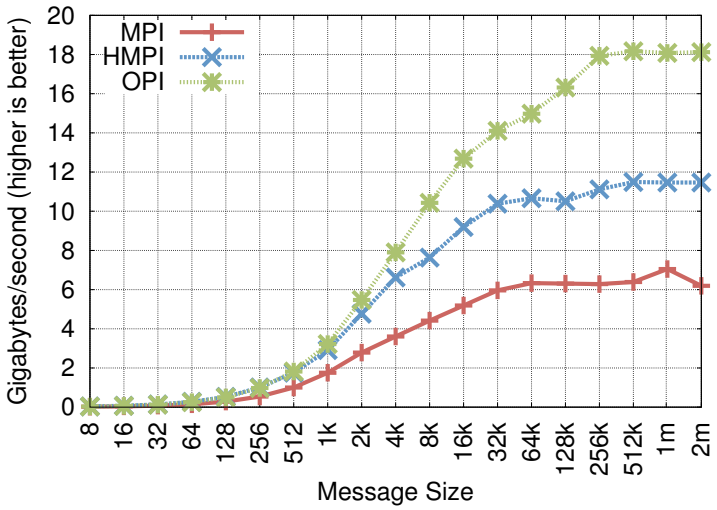
We designed OPBench to allow us to independently benchmark the pack, communication, unpack, and a simulated application computation phase. OPBench implements a simple nearest-neighbor stencil, performing the following steps:

(1) Computation time is simulated and measured by performing a simple calculation on each of the elements of an array.
(2) A pack loop copies the data from the 'application' data structure to a communication buffer.
(3) The communication buffers are exchanged between two ranks.
(4) An unpack loop copies the data from the received communication buffer back to the application data structure.

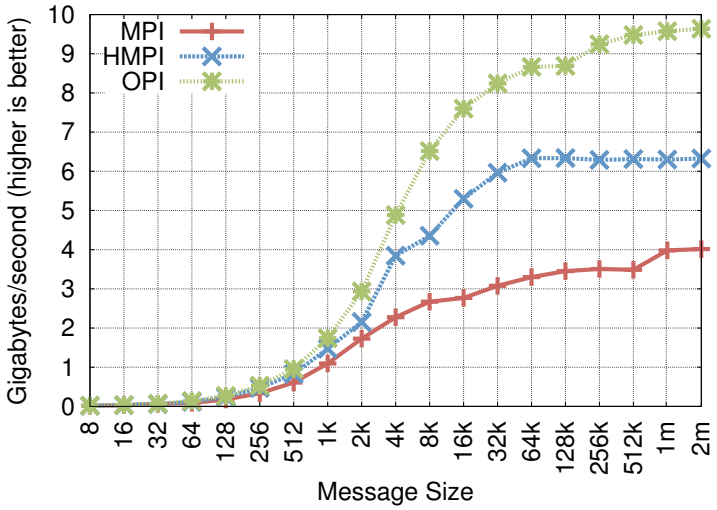
For each iteration of the benchmark we perform step 1 once, then repeat steps 2-4 four times to simulate multiple neighbors. We ran our benchmark in two configurations on Sandy Bridge: (i) ranks are located on different cores within the same processor (socket) and (ii) ranks are located on different processors (sockets). Each data point in the results is an average of 5,000 benchmark iterations, with timings acquired from both ranks. For OPI, a call to `OPI_Alloc()` is included in the pack phase, and a call to `OPI_Free()` in the unpack phase. For all of our OPBench experiments, we chose to show the combined communication and unpack phases (the sum of the time taken by steps 3 and 4 of OPBench) to cover communication time incurred by both message and ownership passing, highlighting the overall differences.

Figure 4.10 shows the bandwidth achieved when communicating within and across processors (sockets) on Sandy Bridge. Within the same socket, OPI (18 gB/sec) significantly improves on HMPI (11.5 gB/sec) and MPI (6.25 gB/sec). For small messages, we see little benefit from OPI: up to one cache line, HMPI and OPI have identical performance. From there, the gap slowly widens as message passing becomes more expensive. As message size grows, OPI performance is bounded only by the memory bandwidth achieved during the unpack phase, where data is streamed directly from the sender with no copying. On the other hand, HMPI and MPI are making one or two copies of the data during the communication phase.

Figure 4.11 shows the same benchmark results for Blue Gene/Q, which has only one processor (socket). The overall picture is similar to the same-socket Sandy Bridge results,



(a) Intra-socket



(b) Inter-socket

FIGURE 4.10. Bandwidth for OPBench communication + unpack phases on Sandy Bridge.

though the peak gap between OPI (18.75 gB/sec) and HMPI (9 gB/sec) is larger. MPI eventually achieves the same data rate as HMPI.

To understand the performance properties of OPI in detail, we measured cache behavior during OPBench execution. Figure 4.12 shows the data cache miss rate during the communication and unpacking phases. The cache miss rate is computed as

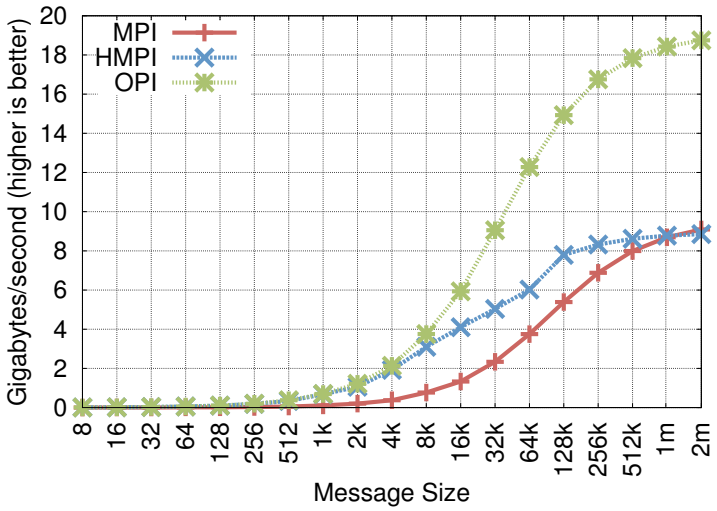
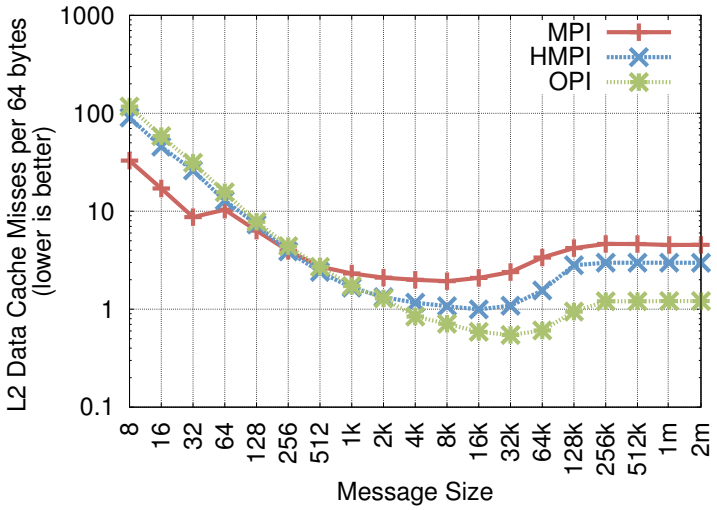


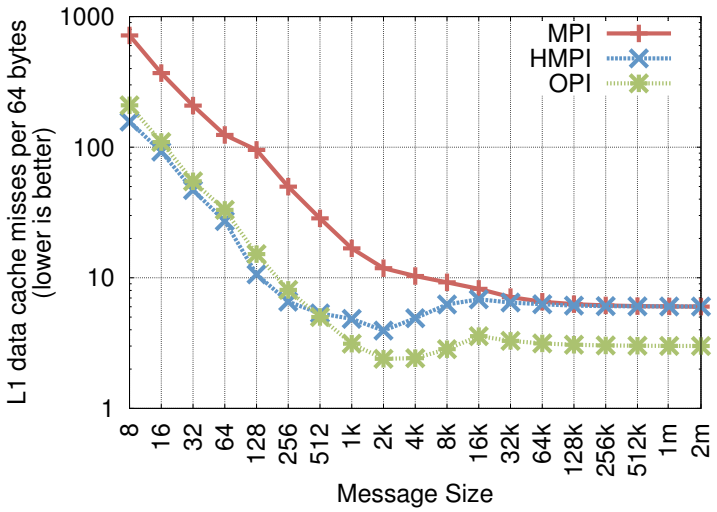
FIGURE 4.11. Bandwidth for OPBench communication + unpack phases on Blue Gene/Q.

$CacheMisses/(MessageSize/64)$, where *CacheMisses* is the total number misses counted by a benchmark run for a specific *MessageSize*. 64 is the size in bytes of a cache line, the atomic unit of memory fetched between cache levels on both Sandy Bridge and Blue Gene/Q. We are interested in showing cache misses related to communication, so we choose the highest level of cache for each system that is exclusive to a core (i.e., not shared with other cores). On Sandy Bridge, we show L2 data cache misses, and L1 data cache misses on Blue Gene/Q.

For small messages, we see a high number of misses per cache line, and that MPI has fewer cache misses on Sandy Bridge. The high miss rate is dominated by the overhead required for MPI message ordering and matching. We believe HMPI and OPI incur more misses due to more cross-rank memory accesses: we use shared memory more often than MPI. That said, the difference is approximately of 2-4 cache misses. As the message size grows, the data transfer overhead dominates, and both systems show similar behavior. Note the logarithmic scale of the figure: for 2 mB message size, MPI is incurring approximately 3.75 times as many misses as OPI on Sandy Bridge. On Blue Gene/Q, the difference is almost exactly 2 times as many misses.



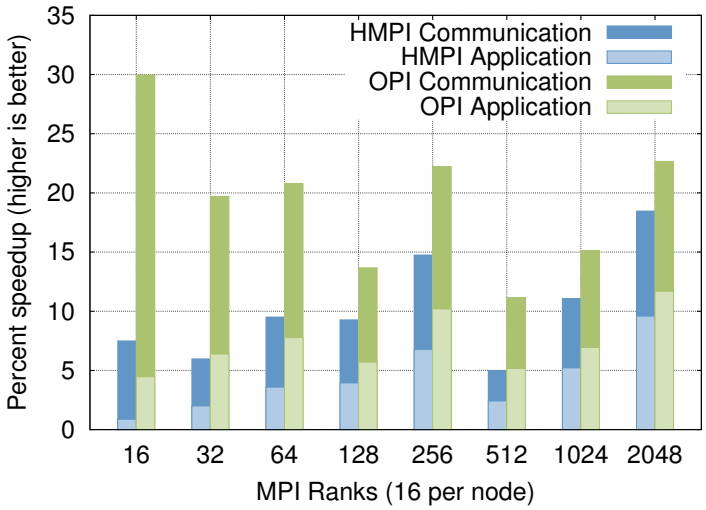
(a) Sandy Bridge Same-socket L2



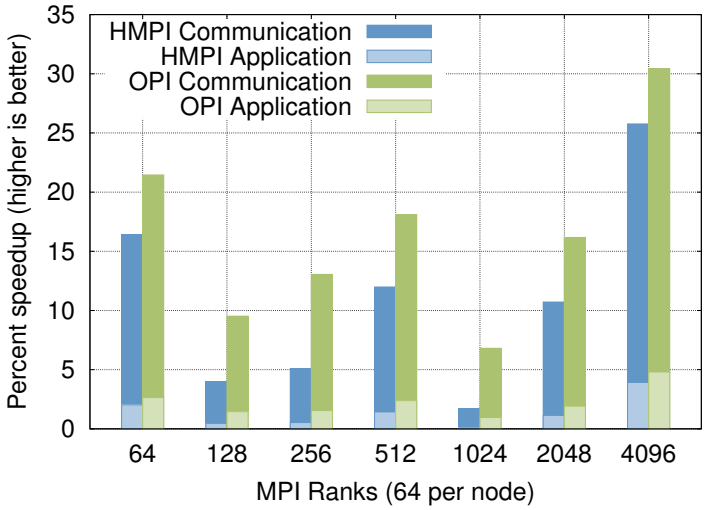
(b) Blue Gene/Q L1

FIGURE 4.12. Cache miss rate for OPBench communicate + unpack phases per 64 bytes (cache line) of message data.

4.7.2. MiniMD. For a description of MiniMD, see Section 3.4.1. The primary communication pattern is a 3D, 6-point nearest neighbor exchange performed twice per work iteration. We transformed the nearest neighbor exchanges to use ownership passing in the same manner as was done for the two-dimensional stencil example described in Section 4.4, modifying the code by hand. Appendix A contains code lists for the original MPI



(a) Sandy Bridge



(b) Blue Gene/Q

FIGURE 4.13. MiniMD speedup using ownership passing (OPI) and HMPI, relative to MPI.

version and the ownership passing version. All other communication was left unchanged, and uses HMPI.

Just like the results from the previous chapter, (Section 3.4.1), we ran 2,500 iterations and used weak scaling with approximately 1,000 atoms. Performance results are shown in

Figure 4.13. Computation of forces between atoms dominates execution time, so optimizing communication has a smaller affect on overall application time. When the communication time alone is considered, significant speedups are observed: 11.2-30.0% on Sandy Bridge and 6.8-30.4% on Blue Gene/Q. Application time speedups range from 4.5-11.7% and 1.0-4.8% for the respective systems.

4.7.3. Fast Fourier Transform. The Fast Fourier Transform (FFT) is among the most important operations in use today. Numerous algorithms and parallel applications use FFTs in their core computations [**29**, **47**]. A one-dimensional FFT transforms a one-dimensional array of N complex numbers from real space to N complex numbers in frequency space. Such a one-dimensional FFT can be expressed in terms of multi-dimensional FFTs with additional application of *twiddle factors* [**15**, **76**]. A multi-dimensional FFT with d dimensions can be computed by applying one-dimensional FFTs along all d axes. Multi-dimensional FFTs are very important in practice; image analysis often requires two-dimensional FFTs and transformations in real-space require three-dimensional FFTs [**29**, **47**].

We perform our experiments using a two-dimensional FFT kernel known as FFT2D. The original 2D FFT code is implemented using MPI and transforms an $N_x \times N_y$ domain. The initial array is stored in x-major order and distributed in y-dimension such that each process has N_x/P y-pencils. The steps to perform the 2D FFT are:

(1) Perform N_x/P 1D FFTs in y-dimension (N_y elements each).
(2) Serialize the array into a buffer for the all-to-all.
(3) Perform a global all-to-all.
(4) De-serialize the array to be contiguous in the x-dimension (each process now has N_y/P x-pencils).
(5) Perform N_y/P 1D FFTs in the x-dimension (N_x elements each).
(6) Serialize the array into a send buffer for the all-to-all.
(7) Perform a global all-to-all.
(8) De-serialize the array into its original layout.

			% Local	
Ranks	Problem Size	Message Size	16 Ranks/Node	64 Ranks/Node
16	$1,024^2$	65,536	100.00%	100.00%
64	$2,048^2$	16,384	25.00%	100.00%
256	$4,096^2$	4,096	6.25%	25.00%
1,024	$8,192^2$	1,024	1.56%	6.25%
4,096	$16,384^2$	256	0.39%	1.56%

FIGURE 4.14. FFT2D message statistics. Message size is the number of bytes sent from one rank to another during the all-to-all exchange.

The all-to-all communications in steps 3 and 6 make 2D FFTs an interesting application for ownership passing. We perform the all-to-all ownership passing transformation described in Section 4.5. Appendix B contains code listings for the original MPI version and the ownership passing version of the communication phase (steps 2-4 and 6-8). Each sender has one memory pool, from which it allocates and packs one buffer for each other rank. As data for each rank is packed, ownership is transferred to the receiver. Buffers are unpacked as ownership control arrives from each other rank.

We experimented with FFT2D using weak scaling, with 65,536 complex double-precision numbers stored on each rank. The overall problem size is determined by the formula $N^2 = NE * NP$, where N is the problem size input to FFT2D (dimension of NxN matrix), NE is the number of elements per rank, and NP is the number of ranks. FFT2D has the further requirement that N be an integer multiple of NP, limiting the number of configurations we can run. Figure 4.14 gives statistics on our configuration of FFT2D over a varying number of MPI ranks. Since the primary communication pattern is all-to-all, and the problem size per rank is fixed, the amount of data sent per message to each other rank decreases as the number of ranks grows. Likewise, the percentage of local communication decreases as a function of the number of ranks.

Figure 4.15 shows the results, with communication time speedups of up to 54% on Sandy Bridge and 37% on Blue Gene/Q. Not surprisingly, these are the single-node results, where all communication is local. As the number of ranks (and nodes) increases, FFT2D is highly network bound, limiting the impact of our intra-node optimization.

We see an interesting phenomenon in the Sandy Bridge results: HMPI eventually offers greater speedups as the rank increases. This is effect is not directly due to the rank count; rather it is due to the decreasing message size. Cross referencing message sizes from the table in Figure 4.14 with the bandwidth charts in Figures 4.10 and 4.11 yields this insight. For small messages, ownership passing provides no speedup as the cost of additional buffer management exceeds the cost of HMPI's data transfer protocols. We see larger speedups for OPI at lower node counts due to the fact that messages are larger and ownership passing provides greater benefit. An alternative that might favor ownership passing over the weak scaling shown might be to fix the message size and grow the overall problem size accordingly (weak weak scaling).

The Blue Gene/Q results show some weaknesses in the HMPI design. The results are good within a single node, but scaling to more nodes results in reduced performance. HMPI adds some overheads to inter-node communication that have proven difficult to eliminate, particularly on Blue Gene/Q. A processor that is relatively slow compared to its memory and network emphasizes the issue. Each send and receive operation requires additional code to check whether it should be handled locally or not. For non-blocking operations (`Isend()` and `Irecv()`), we have to ensure progress is made in both the underlying MPI and in HMPI. To do that, we cannot block inside MPI, and must alternately poll between the two messaging layers. Thus, it is difficult to compete with a single library that can execute a much tighter polling loop. These are mainly limitations of HMPI's layered library approach. We expect that fully integrating our work into an existing MPI would largely solve these problems.

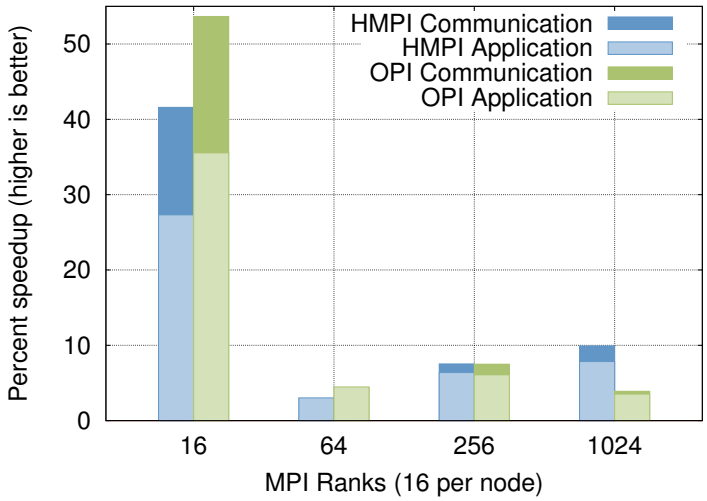
As an aside, our work on Blue Gene/Q led to the discovery that the time to execute correctly predicted branches varies from 3 to 6 cycles depending on the address of the branch instruction and the address of its target. HMPI is largely branch-heavy code, so it is particularly susceptible to this problem. On microbenchmarks, we found as much as 15% additional latency that would vary depending on the existence of functions that are never called or if-statements whose bodies are never executed. Carefully inserting no-op

instructions to adjust the location of branches and their targets can help to improve performance, but the process must be repeated any time the HMPI code is changed. The fact that the Blue Gene/Q processor has a relatively small L1 instruction cache (16kB) compounds this problem, as we had to limit the amount of function inlining and increase the number of function calls (branches) to avoid chronic instruction cache spills. A technique that helps to hide the performance penalty in applications is running multiple threads per core. Blue Gene/Q has hardware support for four-way Simultaneous MultiThreading (SMT). Running 64 ranks per node helps limit the impact of the branch latency problem. Optimizing to such a detailed level of performance has proven consistently difficult on Blue Gene/Q.

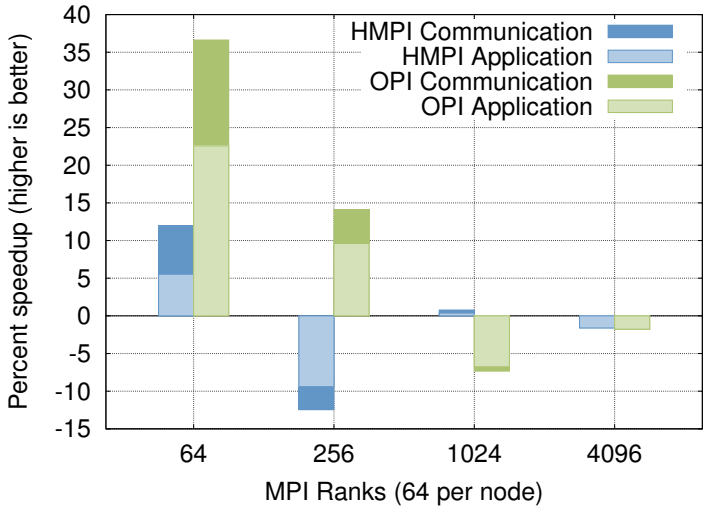
4.8. Related Work

Acknowledging the value of sharing memory between MPI ranks, the MPI Forum introduced shared memory windows in MPI version 3 [**33**, **34**, **66**]. Those windows allow creation of shared memory regions for direct sharing of data. However, the programmer needs to distinguish between on- and off-node communication explicitly and encode either direct data access (on-node) or message passing (off-node) in the application. While MPI offers a mechanism to query the node topology to distinguish between the two, the resulting program code is still rather complex and hard to maintain. Furthermore, shared memory windows aren't guaranteed to be mapped to the same address on all ranks. Pointers into shared memory windows aren't necessarily valid across ranks without doing error-prone address translations. Our ownership passing extension provides similar functionality with much less complexity by maintaining the familiar two-sided send—receive pattern and hiding differences between inter- and intra-node communication.

The DASH operating system used an interesting message passing design for communication [**90**]. Instead of copying data, virtual memory was leveraged to simply remap the message from one virtual memory location to another. Although called message passing, this is effectively ownership passing implemented and enforced using the virtual memory system. It appears that XPMEM [**72**, **88**] could be used to implement the same technique in a more modern context. The downside to this approach is that it requires special support



(a) Sandy Bridge



(b) Blue Gene/Q

FIGURE 4.15. 2D FFT speedup using ownership passing and HMPI, relative to MPI. Weak scaling with 65,536 complex elements per rank was used. Application times include communication time.

from the operating system that isn't available on production HPC systems, limiting the availability and portability of this approach.

The idea of ownership passing draws upon techniques that can be found in Distributed Shared Memory (DSM) systems [77] and early cache coherence protocols [22]. In either

case, *ownership* is defined in the same manner—only one process is entitled to read or write a particular block of memory (e.g., a page or cache line). In this work, we use the concept of ownership to present a clean interface to improved shared memory performance in the context of MPI's distributed memory model.

The Generic Message Passing Framework **[53, 54]** implements a message passing interface for C++ that is reminiscent of MPI, with an extension for doing ownership passing using the `auto_ptr` (now deprecated in favor of `unique_ptr` **[82]**) reference counting pointer class. Our approach to ownership passing is very similar, but integrates with the existing MPI environment, making it possible to incorporate ownership passing into legacy applications written in Fortran, C, and C++.

Multi-Version Variables (MVVs) **[21**, §8] are a language extension to Co-Array Fortran for supporting a producer-consumer communication channel. A memory pool concept similar to our own is used to provide a form of streaming message passing in a PGAS language. Though similar, our work describes a path for modifying legacy MPI applications for improved performance on shared-memory hardware.

Ownership passing has been used to speed up other parallel programming frameworks such as the actor-based framework *ActorFoundry* **[68]**. Significant performance benefits have been demonstrated in this context. However, C or Fortran with MPI codes have a more complex structure than *ActorFoundry* and complete static analysis is thus not always possible.

4.9. Conclusion

We have shown how the principle of ownership passing can be used with MPI applications in order to utilize shared memory (multi-core) hardware more efficiently. This principle is often used implicitly in cache-coherency protocols; we have extended it with a software interface to be used explicitly. Our Ownership Passing Interface (OPI) is a simple extension to MPI and keeps MPI's ease of programming and abstraction (as opposed to shared memory programming with critical sections) while providing true *zero-touch* intra-node communication.

We addressed the challenge of returning the buffers by using several pooling techniques. Our lock-free receiver pooling technique shows best results for practical applications where messages (buffers) are often passed symmetrically between MPI ranks.

Our interface allows the porting of legacy MPI applications to support 'fat' shared memory nodes with simple transformations. Our examples show that the transformation is indeed simple and straightforward to apply.

Our performance studies with microbenchmarks and real applications show that ownership passing is an effective technique for achieving better performance on shared memory hardware. Communication time speedups of up to 30% in a molecular dynamics application and 54% are realized in a two-dimensional FFT code.

5

Shared Heap Techniques

5.1. Introduction

In this chapter, we go beyond message passing and even ownership passing to explore techniques for communication using a shared heap, starting from the distributed memory programming model as a base. In previous chapters, we remained close to the distributed memory abstraction. Here we adopt a truly hybrid approach in which explicit message passing is replaced entirely by shared memory techniques.

First, we take a look at how to replace message-based synchronization with thread-based (in this case, actually process-based) synchronization like locks, semaphores, and other atomic shared memory operations. Given that ownership passing has eliminated the data transfer phase from ownership passing, this step eliminates (or replaces) the matching

phase (see Chapter 3 for discussion of matching and data transfer phases in MPI). The result is a hybrid of shared and distributed memory models: shared memory is used for synchronization, while communication follows the common pack-communicate-unpack pattern found in distributed memory programming. MPI continues to provide process startup and management as well as inter-node communication. Replacing MPI matching with shared memory synchronization allows us to compare the two approaches and their performance characteristics.

Second, we look at how we can hybridize the pack-communicate-unpack pattern by fusing the pack and unpack code into one operation, leaving only a direct data transfer without any data serialization. Packing and unpacking is often expressed as simple loops iterating over arrays, serializing and deserializing into contiguous buffers. By combining the statements from the two loop bodies, we can synthesize a fused loop that eliminates serialization (and intermediate communication buffers), moving data directly from its source location on one rank to its destination on another.

We demonstrate these techniques by using the MiniMD and FFT2D codes shown in previous chapters as case studies for step by step code transformations. In MiniMD, the structure of the pack and unpack loops impact whether the fused loop should be placed on the sender or receiver. Experimental results demonstrate the effects of these shared memory techniques on performance.

The techniques in this chapter demonstrate how Hybrid Message Passing Interface (HMPI) can be used as a platform for aggressive, application-specific performance optimizations in a shared memory environment. The key contributions are the following:

(1) A technique for replacing ownership passing (or message passing) calls with shared memory synchronization.
(2) Loop fusion, a technique for eliminating serialization completely from the common pack-communicate-unpack pattern found in distributed memory model programs.
(3) Performance analysis of these techniques on an x86 Sandy Bridge system.

```
for(i = 0; i < nSwap; i++) {
    PackForward(SendBuf, SendNum[i], SendList[i]);

    MPI_Isend(&SendBuf, SendProc[i], &SendReq);
    MPI_Recv(&RecvBuf, RecvProc[i]);

    UnpackForward(RecvBuf, RecvNum[i], FirstRecv[i]);

    MPI_Wait(&SendReq);
}
```

FIGURE 5.1. MiniMD forward communication phase.

5.2. Case Study: MiniMD

As described before, MiniMD is a molecular dynamics mini-application based on the more complex LAMMPS **[48, 75]** MD code and is part of the Mantevo **[31, 49]** mini-application suite. The movement of atoms is computed in a series of time steps over a 3D space decomposed into a processor grid. In each time step, there are two communication phases to exchange atom position (forward phase) and force (reverse phase) data with neighboring ranks. A rank may have anywhere from zero (MPI configured for only one rank) to six neighbors that it communicates with during these phases.

In this section, we transform the two communication phases to explore how the shared heap can be used to implement different communication techniques and arrive at some performance optimizations. Code examples are shown only for the forward phase here; complete examples for both the forward and reverse phases for all of the transformations in this chapter can be found in Appendix A. We start from the MPI or HMPI version (they are the same) of MiniMD. The OPI version could be used, but the intra-node and inter-node communication paths will be separated into. We will choose MPI for the inter-node communication path since there is no benefit to using OPI in this case. The intra-node path will be transformed to use shared memory directly. Figure 5.1 shows pseudo-code of the forward communication phase we are interested in, which consists of a simple pack-communicate-unpack loop. There are several things to observe:

```
void PackForward(double* buf, int n, int* list)
{
    for(i = 0, m = 0; i < n; i++) {
        int j = list[i];
        buf[m++] = atom.x[j][0];
        buf[m++] = atom.x[j][1];
        buf[m++] = atom.x[j][2];
    }
}

void UnpackForward(double* buf, int n, int first)
{
    for(i = 0, m = 0, k = first; i < n; i++, k++) {
        atom.x[k][0] = buf[m++];
        atom.x[k][1] = buf[m++];
        atom.x[k][2] = buf[m++];
    }
}
```

FIGURE 5.2. MiniMD pack and unpack loops.

- The same send and receive buffers are reused in each iteration. That is, exactly one send and one receive buffer is utilized in either communication phase.
- The neighbor a rank sends data to is not necessarily the same neighbor it receives data from in any particular iteration of the communication.
- Data flow is reversed in the reverse communication phase. Each rank receives data from the rank it sent data to in the forward phase, and vice versa.

The pack and unpack routines are an important part of later code transformation stages. Their original form is shown in Figure 5.2. Notice that the `PackForward()` routine uses a secondary index list.[1] That is, instead of accessing data from the main atom list directly, the index in the atom list is looked up in a secondary index array before accessing the atom for packing or unpacking. Since atoms move around within the domain, the accesses into the primary atom list are effectively random and evolve over time. There is little to no spatial locality. This is a critical component of this code, and is important in later code transformation stages. `UnpackForward()` accesses all data (both the atom list and serialized buffer) sequentially.

[1]The `UnpackReverse()` routine uses the same index list, see Appendix A.

```
typedef struct
{
    void* Buffer;

    signal_t FwdSignal[MaxNeighbors];
    signal_t RevSignal[MaxNeighbors];
} GlobalData;

GlobalData* gData;
```

FIGURE 5.3. Node-wide data structure for shared locks and variables in MiniMD.

5.2.1. Shared Locks and Variables. In Chapter 4, we used a restricted model in which exactly one rank has access to a particular memory region at a time. This model was specifically chosen to preserve that aspect of distributed memory: a rank (process) has sole access to its local memory, and all communication to remote memory is explicit. Now, we relax the ownership passing model to allow any rank to access any memory within the same node at any time. Indeed, this is the actual semantics of the shared memory heap and shared memory in general. We now have many communication tools available: any sort of shared memory synchronization primitives (e.g., pthreads, built-in atomic functions provided by compilers, native atomic instructions), ownership passing (OPI), and message passing (MPI).

Our first transformation replaces the MPI communication shown Figure 5.1 with shared memory locks and variables. One rank per node allocates a region of data to be shared (just by using `malloc()`), then shares the address of the data with other ranks on the node. Sharing pointers only to ranks in the same node is somewhat tricky; MPI versions 1 and 2 **[65]** have no such functionality. MPI version 3.0 **[66]** introduces a new routine that is useful this purpose: `MPI_Comm_split_type()`. This routine allows an existing communicator to be split into smaller communicators based on some *type* attribute associated with each rank. MPI v3.0, defines one type attribute: `MPI_COMM_TYPE_SHARED`, which groups ranks connected by shared memory into their own communicator. Finally, we can share pointers and other data among ranks in a node by using standard MPI collective operations such as broadcast.

Forward Phase

```
for(i = 0; i < nSwap; i++) {
    HMPI_Comm_node_rank(RecvProc[i], &RecvNodeRank);

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Irecv(&RecvBuf, RecvProc[i], &RecvReq);
    } else {
        gData[NodeRank].Buffer = RecvBuf;
        SignalSet(&gData[NodeRank].FwdSignal[i]);
    }

    HMPI_Comm_node_rank(SendProc[i], &SendNodeRank);

    if(SendNodeRank == MPI_UNDEFINED) {
        PackForward(SendBuf, SendNum[i], SendList[i]);
        MPI_Send(&SendBuf, SendProc[i]);
    } else {
        SignalWait(&gData[SendNodeRank].FwdSignal[i], 1);
        PackForward(gData[SendNodeRank.Buffer,
                    SendNum[i], SendList[i]);
        SignalClear(&gData[RecvNodeRank].FwdSignal[i]);
    }

    if(RecvNodeRank == MPI_UNDEFINED)
        MPI_Wait(&RecvReq);
    else
        SignalWait(&gData[NodeRank].FwdSignal[i], 0);

    UnpackForward(RecvBuf, RecvNum[i], FirstRecv[i]);
}
```

FIGURE 5.4. MiniMD forward communication phase transformed to use global (node-wide) locks and variables.

Figure 5.3 shows the data structures and global variables necessary for communicating via shared memory. We define a C structure type containing all of the variables that need to be shared. During program initialization, one rank on each node allocates an array of `GlobalData` structure elements, enough for one per rank in the node. The base address of this array is broadcast to ranks within the same node. Each rank stores a pointer to this data in the `gData` global variable. For MiniMD, the data structure consists of a pointer for communicating the address of packed buffers (messages) and locks for signaling to each neighbor when the buffer pointer is ready for them to access.

We put this shared data structure to use in Figure 5.4, which shows the transformed forward communication phase. All MPI communication is replaced by atomic signals, and a single buffer pointer (per rank) is used to transfer data. This transformation maintains the same pack-communicate-unpack pattern commonly used in distributed memory programming, but makes use of shared memory for communication.

We define some of the routines used in this code. `HMPI_Comm_node_rank()` is an HMPI specific extension that translates a rank number in one communicator (e.g., `MPI_COMM_WORLD` to a rank identifier within the node. Using MPI-3, this can be implemented using `MPI_Comm_split_type()` and then `MPI_Group_translate()` with the resulting communicator representing the local shared memory node. `HMPI_Comm_node_rank()` returns `MPI_UNDEFINED` if given a rank that is not located on the same node. Thus, we can use it to switch between local and remote ranks, and use the translated rank identifier as an index into shared arrays.

The signal routines are wrappers to atomic operations on an integer. `SignalSet()` writes 1 to the variable, while `SignalClear()` writes 0. `SignalWait()` spins until the variable is equal to some value. Furthermore, these atomic operations enforce a memory fence that prevent reads or writes from being moved across the signal routines.

Now for observations on the code structure. The code is noticeably more complex than the original MPI version. We have separated the intra-node communication path from the inter-node path, and use different communication mechanisms in either case. Increased code complexity is a downside to mixed shared and distributed memory programming. Our OPI design from Chapter 4 wraps this sort of flow control inside the give and take operations, hiding the complexity from the application.

Other than introducing complexity to the application code instead of OPI, this code is almost the same as ownership passing. One other critical feature of OPI is missing from this code: memory pools. Here, the sender must wait for the receiver to make its buffer available before packing data. Due to this additional synchronization, the performance of this code is 3.5-15.5% slower on Sandy Bridge. OPI's memory pool functionality avoids the need for synchronization when acquiring a buffer before it is used (packed into). Likewise,

any need for synchronization when releasing a buffer is avoided. In doing this transformation, we found that memory pools not only reduce the cost of `malloc()` and `free()`, they reduce the need for synchronization.

We deliberately chose to structure the code so that the sender packs into the receiver's buffer. The receiver makes its buffer available, waits for the sender, and then unpacks. A more intuitive approach might be to pack into the local send buffer, and then to make that buffer available to the receiver. We took that approach for the reverse phase (see Appendix A). In practice, either way results in the same performance. In the next stage of the transformation, this decision is important due to the use of a secondary index array in the pack loop.

5.2.2. Loop Fusion. Now we can begin to work on transformations to improve performance. In this section, we *fuse* the pack and unpack loops into one loop. Although loop fusion itself is not a new idea [**11**, **44**], to our knowledge this its first application to MPI-based applications.

Notice that the original pack and unpack loops back in Figure 5.2 serialize sequentially into the communication buffer and then deserialize sequentially back out again. We can align the statement that assigns a value from the sender's data structures to a particular element of the communication buffer with the corresponding statement that assigns that value from the communication buffer into the receiver's data structures. In fact, we can see that these statements are both enclosed in loops with the same number of iterations, which corresponds to the number of elements stored in the communication buffer.

In other words, what we have is two loops. One assigns values to a communication buffer. The other assigns those same values to some other data structure. In a sense, the transfer looks like this:

```
Buffer = SenderDataStructure; //Pack loop
RecverDataStructure = Buffer; //Unpack loop
```

By fusing the statements in the pack loop with the statements in the unpack loop, we transform the above statements into the following:

```
void ForwardFusion(double** buf, int n, int* list, int first)
{
    for(i = 0, k = first; i < n; i++, k++) {
        int j = list[i];
        buf[k][0] = atom.x[j][0];
        buf[k][1] = atom.x[j][1];
        buf[k][2] = atom.x[j][2];
    }
}
```

FIGURE 5.5. MiniMD loop fusion routine.

```
RecverDataStructure = SenderDataStructure; //Fused loop
```

Serialization and deserialization into a communication buffer are eliminated, removing the need for any communication buffers. We expect to see a performance benefit because there are fewer assignment (memory copy) operations. There is also a reduction in memory usage since the need for a temporary (communication) buffer has been eliminated. What is left is a *fused loop* operation that is equivalent to the former pack-communicate-unpack pattern, but is more efficient.

The resulting code is deceptively simple given what it represents: the motion of code from one rank to another and subsequent combining of loop bodies. This sort of code motion can be strange to think about, since all we are really doing is rearranging statements in a single program. In the context of parallelism, however, this rearrangement crosses the boundaries of execution contexts. Of course, such a transformation is not possible in strictly distributed memory; all data involved must be directly accessible by the rank executing the fused loop (i.e., located in shared memory).

Figure 5.5 shows the result of applying loop fusion to the forward pack and unpack routines. Notice that the `outbuf` variable is now a double pointer. It now represents a remote rank's main atom array, rather than a communication buffer. The remaining arguments required by the fused loop routine form the union of the arguments of the pack and unpack loops. The `list` argument (secondary index array) must be provided by the sender, while the `first` argument is provided by the receiver.

Some changes to the global variables and the communication phase are needed as well. The `ForwardFusion()` routine is executed by the sender in place of the old pack routine. The unpack routine is no longer necessary for intra-node communication. As mentioned above, the `first` argument to the fused loop must be provided by the receiver. We simply add another variable to our `GlobalData` structure for this purpose (see Appendix A for the code). Figure 5.6 shows MiniMD's forward communication phase transformed for loop fusion. The inter-node communication code remains unchanged, only the intra-node path is modified for loop fusion.

Now we can explain why it was important for the sender to pack into the receiver's buffer in the previous section, and why we chose to place the fused loop on the sender. The reason is the secondary index array used in the pack loop. Normally, placing the fused loop on the receiver is ideal, as it results in remote data being pulled in (read) to the receiving processor core's cache hierarchy. Placing the fused loop on the sender results in greater cache pollution, and the newly transferred data is almost certainly not going to be present in the receiver's cache. However, the secondary index array requires more accesses into the sender's data structures. To find an element to be packed, it first requires a lookup into the secondary array. While these lookups are sequential, they cause the accesses into the main atom list to be effectively random, making poor use of spatial locality. Thus, we found that it is better to place the fused loop on the sender where the secondary array is local, and all or part of the main atom list may already be in local cache.

In the case of the reverse communication phase, this same secondary index array is accessed in the unpack routine. In that case, we placed the fused loop on the receiver to make the best use of cache locality.

Performance is improved after applying the loop fusion transformation to both communication phases. Figure 5.7 shows results from the Sandy Bridge system. Here, we compare performance of loop fusion directly to ownership passing, which is the fastest version of MiniMD shown so far. We see that loop fusion consistently improves performance even further. Communication time speedups range from 4.7% to 9.4% over using

Forward Phase

```
for(i = 0; i < nSwap; i++) {
    HMPI_Comm_node_rank(RecvProc[i], &RecvNodeRank);

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Irecv(&RecvBuf, RecvProc[i], &RecvReq);
    } else {
        gData[NodeRank].Buffer = atom.x;
        gData[NodeRank].First = FirstRecv[i];
        SignalSet(&gData[NodeRank].FwdSignal[i]);
    }

    HMPI_Comm_node_rank(SendProc[i], &SendNodeRank);

    if(SendNodeRank == MPI_UNDEFINED) {
        PackForward(SendBuf, SendNum[i], SendList[i]);
        MPI_Send(&SendBuf, SendProc[i]);
    } else {
        SignalWait(&gData[SendNodeRank].FwdSignal[i], 1);
        ForwardFusion(gData[SendNodeRank].Buffer,
                      SendNum[i], SendList[i],
                      gData[SendNodeRank].First);
        SignalClear(&gData[SendNodeRank].FwdSignal[i]);
    }

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Wait(&RecvReq);
        UnpackForward(RecvBuf, RecvNum[i], FirstRecv[i]);
    } else {
        SignalWait(&gData[NodeRank].FwdSignal[i], 0);
    }
}
```

FIGURE 5.6. MiniMD forward communication phase transformed to use loop fusion.

OPI. Although the transformations required for loop fusion result in increased code complexity, the improvement is enough to be worthwhile.

5.3. Case Study: FFT2D

The FFT2D code computes a two-dimensional FFT on a matrix of elements distributed across multiple ranks. A one-dimensional FFT transforms an array of N complex numbers from real space to N complex numbers in frequency space. A two-dimensional FFT can be

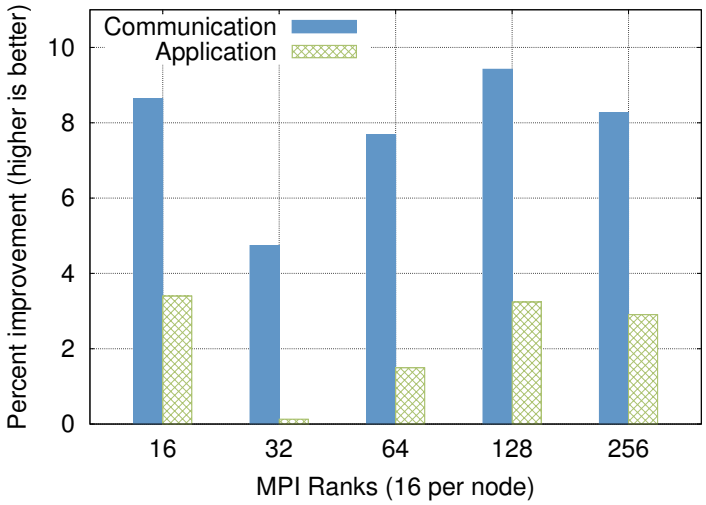


FIGURE 5.7. MiniMD speedup using loop fusion relative to Ownership Passing Interface (OPI).

constructed by applying N one-dimensional FFTs in one dimension, then applying another N one-dimensional FFTs in the other dimension. In the FFT2D code, apply 1D FFTs in the x dimension, transpose the 2D matrix of complex numbers using an all-to-all communication pattern, then repeat for the y dimension. Section 4.7.3 explains the overall code in detail; here we focus only on the communication.

Transposing a 2D matrix distributed across many ranks results in fairly complex array indexing, so we illustrate the process in Figure 5.8. Each rank holds a NxN/P region (pencil) of the NxN matrix, where N is the number of complex numbers along one dimension (also known as the problem size), and P is the number of ranks. The data on each rank is further broken up into $N/PxN/P$ sized blocks. During the all-to-all, rank i sends its local block j to rank j, who receives the data into its local block i. Packing is required to serialize each block down to a contiguous buffer for communication. When unpacking, we transpose the elements of the block while deserializing the contiguous communication buffer.

Figure 5.9 shows the all-to-all communication process in pseudo-code. As each block is packed, we initiate a non-blocking MPI send. During the unpack loop, we wait for the data from each rank to arrive, then unpack and transpose the block of data from that rank. FFT2D executes the same communication sequence twice for each 2D FFT.

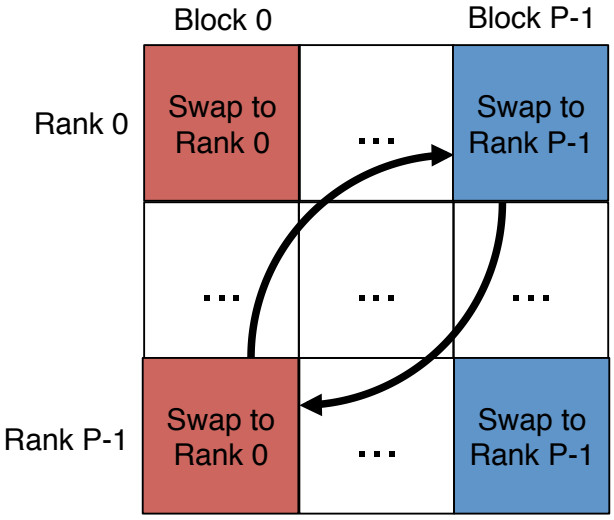


FIGURE 5.8. FFT2D all-to-all matrix transposition. Block i on rank j is copied to block j on rank i. Within the block, the elements are also transposed.

The block pack and unpack routines are shown in Figure 5.10. At first glance they appear very similar, but the unpack routine has an important difference: elements of the block are transposed as they are copied. On the other hand, the pack routine copies elements sequentially in the y direction. Although we chose to express the code this way for clarity when comparing this code to loop fusion version, the inner y loop could be replaced with a `memcpy()` call.

5.3.1. Loop Fusion. In the MiniMD case study, we introduced the concepts for transforming MPI communication to use shared memory locks, and then for fusing the pack and unpack loops. Although FFT2D utilizes an all-to-all communication instead of a nearest neighbor exchange, it still has the same pack-communicate-unpack pattern. We apply the same transformations to arrive at a version of FFT2D modified for loop fusion.

Figure 5.11 shows the global data structures necessary for synchronizing and sharing data. Just as with MiniMD, each rank gets its own instance of the GlobalData structure organized into an array shared within each node.

Since FFT2D communicates in an all-to-all pattern, the necessary synchronization is different from MiniMD. Instead of using the signal primitives as was done in MiniMD, we

```
int BlockSize = ProblemSize / NumRanks;

// Pack & Send loop
for(rank = 0; rank < NumRanks; rank++) {
    SrcBuf = &FFTBuf[rank * BlockSize];
    DestBuf = &SendBuf[rank * BlockSize * BlockSize];

    PackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);

    MPI_Isend(DestBuf, rank, SendReq[rank]);
}

for(rank = 0; rank < NumRanks; rank++)
    MPI_Irecv(&RecvBuf[rank * BlockSize * BlockSize],
              rank, RecvReq[rank]);

MPI_Waitall(NumRanks, SendReq);

// Receive & Unpack loop
for(rank = 0; rank < NumRanks; rank++) {
    MPI_Wait(&RecvReq[rank]);
    SrcBuf = &RecvBuf[rank * BlockSize * BlockSize];
    DestBuf = &FFTBuf[rank * BlockSize];

    UnpackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);
}
```

FIGURE 5.9. MPI version of FFT2D all-to-all communication, forming a distributed matrix transposition.

chose an atomic counter primitive for FFT2D. We implemented the counter using built-in atomic functions first defined by Intel [**17**] and now also supported by the GNU [**16**] and IBM [**39**] C/C++ compilers. Although not standardized and very low level, support by at least these three major compilers makes the built-in atomic functions fairly portable.

The `CounterAdd()` operation is equivalent to atomic fetch-and-add. That is, it reads the prior value of the counter and increments the counter by some amount, all as one atomic operation. `CounterWaitGT()` spins waiting for the counter to be greater than some specified value. In the FFT2D code examples shown here, we use these primitives to create a node-wide barrier synchronization that protects the all-to-all data exchanges.

Figure 5.12 shows the communication code after transforming it to use shared memory synchronization and loop fusion. For completeness, Appendix B lists code transformed

```
void PackBlock(complex_t* DestBuf, complex_t* SrcBuf,
               int BlockSize, int ProblemSize)
{
    for(x = 0; x < BlockSize; x++) {
        for(y = 0; y < BlockSize; y++) {
            DestBuf[x * BlockSize + y] =
                    SrcBuf[x * Problemsize + y];
        }
    }
}

void UnpackBlock(complex_t* DestBuf, complex_t* SrcBuf,
                 int BlockSize, int ProblemSize)
{
    // Transpose while unpacking
    for(x = 0; x < BlockSize; x++) {
        for(y = 0; y < BlockSize; y++) {
            DestBuf[y * ProblemSize + x] =
                    SrcBuf[x * BlockSize + y];
        }
    }
}
```

FIGURE 5.10. Original FFT2D pack and unpack (with transpose) routines.

```
typedef struct
{
    void* Buffer;
    counter_t Counter;
} GlobalData;

GlobalData* gData;
```

FIGURE 5.11. Node-wide data structure for shared locks and variables in FFT2D.

only for using shared memory synchronization. As with MiniMD, the code is more complex due to the introduction of separate control flow for intra-node and inter-node communication. The synchronization scheme is also quite different, as we use atomic counters instead of signals. Although the same signal primitives could be used here, they result in more complex code.

The code for the fused pack and unpack loops is deceptively simple, as seen in Figure 5.13. An important distinction is that `SrcBuf` is no longer expected to be a packed

```
int BlockSize = ProblemSize / NumRanks;

gData[NodeRank].Buffer = FFTBuf;
Counter = CounterAdd(&gData[NodeRank].Counter, 1);

for(rank = 0; rank < NumRanks; rank++) {
    HMPI_Comm_node_rank(rank, &SendNodeRank);

    if(SendNodeRank == MPI_UNDEFINED) {
        SrcBuf = &FFTBuf[rank * BlockSize];
        DestBuf = &SendBuf[rank * BlockSize * BlockSize];

        PackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);
        MPI_Isend(DestBuf, rank, SendReq[rank]);
    } else {
        CounterWaitGT(&gData[SendNodeRank].Counter, Counter);

        SrcBuf = &gData[SendNodeRank].Buffer[MyRank * BlockSize;
        DestBuf = &FFTBuf[rank * BlockSize];

        FuseBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);
    }
}

Counter = CounterAdd(&gData[NodeRank].Counter, 1);

for(t = 0; t < NodeSize; t++)
    CounterWaitGT(&gData[RecvNodeRank].Counter, Counter);

for(rank = 0; rank < NumRanks; rank++)
    MPI_Irecv(&RecvBuf[rank * BlockSize * BlockSize],
              rank, RecvReq[rank]);

MPI_Waitall(NumRanks, SendReq);

for(rank = 0; rank < NumRanks; rank++) {
    HMPI_Comm_node_rank(rank, &RecvNodeRank);

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Wait(&RecvReq[rank]);

        DestBuf = &FFTBuf[rank * BlockSize];
        SrcBuf = &RecvBuf[rank * BlockSize * BlockSize];

        UnpackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);
    }
}
```

FIGURE 5.12. FFT2D all-to-all communication transformed for loop fusion.

```
void FuseBlock(complex_t* DestBuf, complex_t* SrcBuf,
               int BlockSize, int ProblemSize)
{
    for(x = 0; x < BlockSize; x++) {
        for(y = 0; y < BlockSize; y++) {
            DestBuf[y * ProblemSize + x] =
                    SrcBuf[x * ProblemSize + y];
        }
    }
}
```

FIGURE 5.13. FFT2D loop fusion routine.

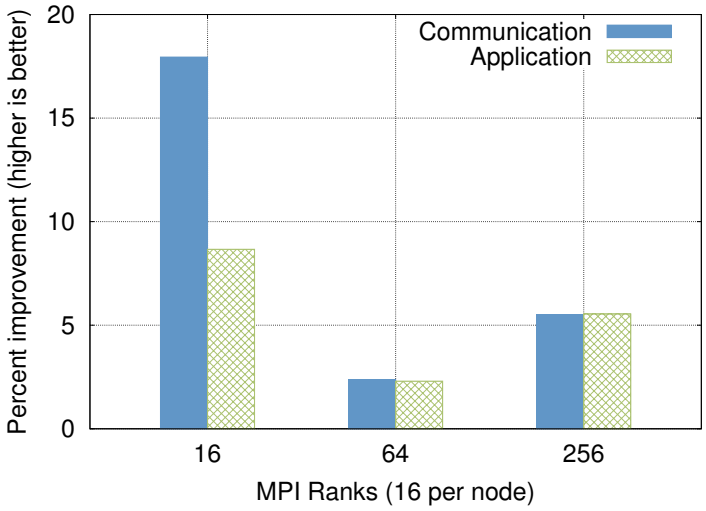


FIGURE 5.14. FFT2D speedup using loop fusion relative to Ownership Passing Interface (OPI).

communication buffer. Instead, it points directly to the source rank's main data structure—just like the fused loop for MiniMD. As a result, the indexing to create the `SrcBuf` pointer in the main communication code has to change. Each rank has to find its data region in the source rank's two-dimensional sub-matrix of complex numbers.

The real complexity of loop fusion is not the fused loop itself, but setting up the data structures, synchronization, and control flow to make it possible. In Figure 5.12, note that we placed the fused loop data transfer as an alternate branch to the inter-node pack and send code, rather than placing with the receive and unpack code. This arrangement reduces the need for additional buffers.

Performance results comparing loop fusion to OPI for FFT2D are shown in Figure 5.14. At more than one node, communication time dominates overall execution time, so communication speedups translate closely to application speedups. Despite the fact that there is very little local communication as the node count increases, we still see speedups due to loop fusion. Of course, loop fusion has the largest effect (18%) when running with a single node, where all communication is local.

5.4. Further Techniques

In this dissertation, and especially in this chapter, we have presented code transformations that were applied by hand. However, this work is part of a larger project to develop a compiler tool for automatically optimizing MPI codes for next-generation high-performance computers. Alongside this work, compiler techniques have been developed to automatically transform codes to use loop fusion [7]. A source-to-source compiler tool enables applications to take advantage of our performance optimizations without manual programming. Not only is programmer effort reduced, the automatic compiler techniques help to hide the increased complexity of separating inter- and intra-node communication that we saw in this chapter.

There are a myriad of ways to implement communication. For example, we could consider more variations where shared memory data structures are protected by node-level MPI collectives like `MPI_Barrier()` for synchronization. We could use OPI to pass pointers to data structures instead of using shared memory, and it would still be possible to utilize the loop fusion technique. Taking the loop fusion concept even further, it may be possible (depending on the application) to eliminate explicit communication phases completely, implicitly sharing data among ranks on a node in the same manner as might be done when using OpenMP or multi-threading. HMPI and the shared heap create many opportunities for utilizing shared memory to optimize communication performance in multi-code compute nodes.

One requirement for applying the transformations shown in this chapter is that the inter- and intra-node communication paths had to be separated. Both MPI and OPI are

designed such that this is not necessary; communication with another rank is the same regardless of whether it is located on the same node or across the network. Separating the communication paths creates an opportunity for optimizing inter-node communication. HMPI could be bypassed to use the underlying MPI directly, since we know the communication is not local to the node. In fact, other than the shared heap, HMPI is not necessary if all intra-node communication is transformed to use shared memory. Furthermore, compiler tools [**19**] exist that can automatically transform MPI calls to use network-specific communication libraries. Such a tool could be used to bypass MPI completely for most (or all) communication, reducing overhead.

5.5. Conclusion

In this chapter, we used case studies of two different application codes, MiniMD and FFT2D, to show different ways in which message passing communication can be replaced with shared memory synchronization and data structures. In doing so, we preserved the existing pack-communicate-unpack pattern while utilizing the shared heap. We found that this first transformation does not necessarily lead to improved performance—MPI as a synchronization tool appears to be sufficient.

Part of the reason for that is the use of communication-specific buffers and data copying. Although the perception is that this is unnecessary work and creates overhead, it reduces the rigidity of inter-process synchronization. A more relaxed synchronization model in turn can lead to greater performance. On the other hand, the loop fusion transformation can aggressively eliminate data serialization and intermediate buffers to provide an overall performance improvement, despite more rigid synchronization. Such optimizations are only increasing in importance as the number of cores per node continues to grow.

We introduced the concept of loop fusion, a technique in which explicit communication and intermediate communication buffers can be replaced by a direct data transfer operation. Although more specific (i.e., limited to loop-based pack-communicate-unpack programming patterns) than ownership passing, loop fusion provides greater performance speedups. This technique demonstrates the potential for achieving maximum shared

memory performance from a distributed memory MPI code. For MiniMD we demonstrated up to a 9.4% increase in communication performance over our own ownership passing optimization, and up to 18% for FFT2D.

In this chapter we explored the foundation of these sorts of truly hybrid ideas that combine distributed and shared memory programming models. Shared memory techniques can be used to replace all explicit MPI communication within a node. We continue to use MPI for its fast and scalable inter-node communication while taking advantage of shared memory within a node for ideal performance in both areas. Supported by automatic compiler techniques to apply our shared memory optimization, we have shown that distributed memory MPI remains an ideal programming model for current and future HPC systems.

6

Conclusions

In this dissertation, we have developed a series of optimization techniques that make MPI , as an example of a distributed memory programming model, more effective in a shared memory hardware environment. Our work is crucial in addressing the challenge of programming for modern and future HPC systems that feature multiple cores per node. As the number of cores per processor increases, more MPI ranks are connected by shared memory rather than a network. The systems used for experimentation in our work both had 16 cores per node, while Blue Gene/Q has four-way SMT support resulting in support for 64 ranks per node. Forecasts for exascale systems [**45**] predict hundreds or thousands of cores per node in the near future.

Chapter 2 introduced our Hybrid Message Passing Interface (HMPI) library, which has a layered design to increase portability and leverage functionality already present in existing MPI libraries. Furthermore, we introduced our user-space implementation of the shared memory heap, which enables shared memory by default across a set of processes within a node. The shared heap is the fundamental idea upon which all performance optimizations in this dissertation are built on. Memory allocated by the standard `malloc()` et al. system routines is accessible by all other participating processes without the need for any address translation or other OS-supported extensions. Maintaining a process-based model means that memory is effectively private until addresses (pointers) are shared with other processes.

In Chapter 3, we showed how message passing can take advantage of a shared heap. First, like other prior work involving kernel extensions, we can transfer data using a single copy instead of the pipelined two-copy approach used in existing MPI libraries. Next we developed two novel protocols that improve performance beyond that of a single memory copy. Our *immediate* protocol actually uses two copies to strategically avoid incurring an extra cache miss, providing lower latency for small messages. For large messages, our *synergistic* protocol allows both the sending and receiving rank rank to perform parts of a single memory copy. Since a single core cannot saturate the memory and cache bandwidth, using two cores to copy data results in higher bandwidth. We demonstrate communication time speedups of up to 26.2% and 46.1% for the MiniMD and LULESH mini-applications, respectively.

Ownership passing extends MPI to allow the transfer of control, or *ownership*, of a communication buffer instead of making a copy. Chapter 4 presented this idea. The shared heap makes the implementation of this idea relatively simple: we merely pass a pointer to a data buffer. We defined new `OPI_Give()` and `OPI_Take()` operations for ownership transfer that also transparently emulate ownership passing across nodes (distributed memory partitions) by falling back on `MPI_Send()` and `MPI_Receive()`. Ownership passing results in a design pattern in which a communication buffer is allocated for each message to be shared. To alleviate the high costs of memory allocation, we developed a

memory pool interface for efficiently managing communication buffers. Communication performance is increased by up to 30% for MiniMD and up to 54% for FFT2D.

Finally, Chapter 5 explored application-level techniques for utilizing shared memory in the context of MPI. First, we showed how explicit MPI communication calls can be replaced by the use of existing shared memory synchronization primitives and shared data structures. Our loop fusion technique transforms code designed using the common pack-communicate-unpack pattern for even greater performance results. We combine the loops forming the pack and unpack phases into a single fused loop, eliminating explicit communication calls and intermediate communication buffers. Experimental results with MiniMD and FFT2D show additional communication time speedups of up to 9.4% and 18% over ownership passing, respectively.

6.1. Future Work

We chose to focus on optimizations for point-to-point message passing in shared memory to show that distributed memory programming models can be effective in shared memory. However, many of our techniques and perhaps more related approaches to exploiting the shared memory heap are applicable to other parallel programming models and inter-process communication in general. Thus, an area of future work would be to explore uses of the shared memory heap, our novel data transfer protocols, and ownership passing in other parallel programming contexts.

For ownership passing (Chapter 4), we chose a strict definition of ownership that allows only one rank to read or write a region of memory at any one time. In Chapter 5 we fully relaxed this memory model to allow any rank to read or write any memory at any time. Work by Mahajan et al. **[61]** has been done utilizing memory models that fall in between these two extremes, maintaining the safety of a distributed private-only memory model while yielding more of the gains possible with fully shared memory. Each buffer is assigned a reference count for tracking access to the buffers, allowing ownership passing with multiple readers and copying data when necessary due to writes. Further work on buffer management for ownership passing could solve the problem of splitting buffers up

into sub-regions with different ownership states. In particular this would be useful for collective communication patterns like scatter and gather that split or combine data to or from multiple ranks. S. Li et al. appears to be extending his work on NUMA-aware shared memory collectives [**56**] to utilize ownership passing.

In Chapter 4, we observed some unwanted overhead due to the HMPI design, which requires polling of both the intra-node HMPI communication layer and the underlying MPI implementation to make communication progress. Integrating our work into existing MPI libraries would solve this problem. Furthermore, we'd like to see the integration of our work into MPI implementations to make it available to the mainstream HPC community.

Chapter 5 introduced application-specific shared memory synchronization techniques as well as loop fusion. We believe further work could uncover techniques for eliminating and/or optimizing data duplicated across neighboring ranks due to spatial problem decomposition (a.k.a. halo zones, ghost cells, and border regions). MiniMD in particular could benefit from this type of work, as its main communication phases consist of duplicating atom position and force information across neighboring ranks. Instead of copying and duplicating that data, the appropriate atom data could be accessed directly.

In general, one of the complaints against an MPI-everywhere approach is that too many MPI ranks results in decomposing problems into portions that are too small for good performance. For example, splitting a 3D space into smaller and smaller pieces results in larger 'surface area' or border regions between MPI ranks: less computation and more communication. The proposed solution is to configure one MPI rank per node, then use OpenMP to utilize all the processor cores within the node. However we believe this approach is flawed. One, we will eventually arrive again at the same problem of over-decomposition as the number of nodes in a single system increases. Two, we believe it is possible to utilize the same decomposition techniques for the MPI+OpenMP approach using some form of ownership passing and/or techniques based on the shared heap. An important direction for future work would be to understand how problems are being decomposed for MPI+OpenMP and carry them back to an MPI-only model.

We have frequently been asked to compare our work to hybrid MPI+OpenMP programming. We have chosen to focus specifically on optimizing a single programming model (distributed memory as defined by MPI) to address the challenges posed by modern and future HPC systems featuring an increasing number of processor cores per node. In the future, it would be very interesting to objectively compare both the programming experience itself, and the relative performance of our approach to existing approaches like MPI+OpenMP and PGAS languages such as Unified Parallel C.

Finally, we believe that MPI+X is indeed the future of large scale, high performance parallel computing. However, most consider 'X' to be another parallel (shared memory) programming model such as OpenMP. Instead, 'X' could represent a compiler or source-to-source translator tool that understands MPI semantics and has the ability to translate MPI-based communication to use techniques best suited for the hardware environment. We have collaborated to develop automatic compiler techniques for transforming MPI codes to use ownership passing [**24**] and loop fusion [**7**]. Mahajan et al. has also worked towards ownership passing supported by a compiler tool based on ROSE [**61**, **78**]. We hope that research along this line continues, enabling development of applications using only one parallel programming model with the support of optimizing compiler tools.

A

MiniMD Code Examples

This appendix consists of a series of pseudo-code figures illustrating the important communication-related code found in MiniMD. There are two communication phases, forward and reverse. In each phase, a rank exchanges data with up to six neighbors. In the reverse phase, the flow of data is reversed. Each rank receives data from the rank it sent data to in the forward phase, and vice versa. We show versions of the communication code with MPI/HMPI (Chapter 3), OPI (Chapter 4), and the shared memory techniques discussed in Chapter 5.

Forward Phase

```
for(i = 0; i < nSwap; i++) {
    PackForward(SendBuf, SendNum[i], SendList[i]);

    MPI_Irecv(RecvBuf, RecvProc[i], &RecvReq);
    MPI_Isend(SendBuf, SendProc[i], &SendReq);

    MPI_Wait(&RecvReq);

    UnpackForward(RecvBuf, RecvNum[i], FirstRecv[i]);

    MPI_Wait(&SendReq);
}
```

Reverse Phase

```
for(i = 0; i < nSwap; i++) {
    PackReverse(SendBuf, RecvNum[i], FirstRecv[i]);

    MPI_Irecv(RecvBuf, SendProc[i], &RecvReq);
    MPI_Isend(SendBuf, RecvProc[i], &SendReq);

    MPI_Wait(&RecvReq);

    UnpackReverse(RecvBuf, SendNum[i], SendList[i]);

    MPI_Wait(&SendReq);
}
```

FIGURE A.1. MiniMD forward and reverse communication phases in (unmodified) MPI form.

Forward Phase

```
for(i = 0; i < nSwap; i++) {
    OPI_Alloc(&SendBuf, SendNum[i]);
    PackForward(SendBuf, SendNum[i], SendList[i]);

    OPI_Itake(&RecvBuf, RecvProc[i], &RecvReq);
    OPI_Igive(&SendBuf, SendProc[i], &SendReq);

    MPI_Wait(&RecvReq);

    UnpackForward(RecvBuf, RecvNum[i], FirstRecv[i]);
    OPI_Free(&RecvBuf);

    MPI_Wait(&SendReq);
}
```

Reverse Phase

```
for(i = 0; i < nSwap; i++) {
    OPI_Alloc(&SendBuf, SendNum[i]);
    PackReverse(SendBuf, RecvNum[i], FirstRecv[i]);

    OPI_Itake(&RecvBuf, SendProc[i], &RecvReq);
    OPI_Igive(&SendBuf, RecvProc[i], &SendReq);

    MPI_Wait(&RecvReq);

    UnpackReverse(RecvBuf, SendNum[i], SendList[i]);
    OPI_Free(&RecvBuf);

    MPI_Wait(&SendReq);
}
```

FIGURE A.2. MiniMD forward and reverse communication phases in OPI form.

Forward Phase

```
void PackForward(double* buf, int n, int* list)
{
    for(i = 0, m = 0; i < n; i++) {
        int j = list[i];
        buf[m++] = atom.x[j][0];
        buf[m++] = atom.x[j][1];
        buf[m++] = atom.x[j][2];
    }
}

void UnpackForward(double* buf, int n, int first)
{
    for(i = 0, m = 0, j = first; i < n; i++, k++) {
        atom.x[k][0] = buf[m++];
        atom.x[k][1] = buf[m++];
        atom.x[k][2] = buf[m++];
    }
}
```

Reverse Phase

```
void PackReverse(double* buf, int n, int first)
{
    for(i = 0, m = 0, j = first; i < n; i++, k++) {
        buf[m++] = atom.f[k][0];
        buf[m++] = atom.f[k][1];
        buf[m++] = atom.f[k][2];
    }

}

void UnpackReverse(double* buf, int n, int* list)
{
    for(i = 0, m = 0; i < n; i++) {
        int j = list[i];
        atom.f[j][0] += buf[m++];
        atom.f[j][1] += buf[m++];
        atom.f[j][2] += buf[m++];
    }
}
```

FIGURE A.3. Unmodified MiniMD pack and unpack loops.

Forward Phase

```
void ForwardFusion(double** buf, int n, int* list, int first)
{
    for(i = 0, k = first; i < n; i++, k++) {
        int j = list[i];
        buf[k][0] = atom.x[j][0];
        buf[k][1] = atom.x[j][1];
        buf[k][2] = atom.x[j][2];
    }
}
```

Reverse Phase

```
void ReverseFusion(double** buf, int n, int* list, int first)
{
    for(i = 0, k = first; i < n; i++, k++) {
        int j = list[i];
        atom.f[j][0] += buf[k][0];
        atom.f[j][1] += buf[k][1];
        atom.f[j][2] += buf[k][2];
    }
}
```

FIGURE A.4. MiniMD loop fusion routines.

```
typedef struct
{
    void* Buffer;
    int* First;

    signal_t FwdSignal[MaxNeighbors];
    signal_t RevSignal[MaxNeighbors];
} GlobalData;

GlobalData* gData;
```

FIGURE A.5. Node-wide data structure for shared locks and variables in MiniMD.

Forward Phase

```
for(i = 0; i < nSwap; i++) {
    PackForward(SendBuf, SendNum[i], SendList[i]);

    HMPI_Comm_node_rank(SendProc[i], &SendNodeRank);

    if(SendNodeRank == MPI_UNDEFINED) {
        MPI_Isend(&SendBuf, SendProc[i], &SendReq);
    } else {
        gData[NodeRank].Buffer = SendBuf;
        SignalSet(&gData[NodeRank].FwdSignal[i]);
    }

    HMPI_Comm_node_rank(RecvProc[i], &RecvNodeRank);

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Recv(&RecvBuf, RecvProc[i], &RecvReq);
        UnpackForward(RecvBuf, RecvNum[i], FirstRecv[i]);
    } else {
        SignalWait(&gData[RecvNodeRank].FwdSignal[i], 1);
        UnpackForward(gData[RecvNodeRank],
                      RecvNum[i], FirstRecv[i]);
        SignalClear(&gData[RecvNodeRank].FwdSignal[i]);
    }

    if(SendNodeRank == MPI_UNDEFINED)
        MPI_Wait(&SendReq);
    else
        SignalWait(&gData[NodeRank].FwdSignal[i], 0);
}
```

FIGURE A.6. MiniMD forward communication phase transformed for shared memory synchronization.

Reverse Phase

```
for(i = 0; i < nSwap; i++) {
    PackReverse(SendBuf, RecvNum[i], FirstRecv[i]);

    HMPI_Comm_node_rank(RecvProc[i], &RecvNodeRank);

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Isend(&SendBuf, RecvProc[i], &SendReq);
    } else {
        gData[NodeRank].Buffer = SendBuf;
        SignalSet(&gData[NodeRank].RevSignal[i]);
    }

    HMPI_Comm_node_rank(SendProc[i], &SendNodeRank);

    if(SendNodeRank == MPI_UNDEFINED) {
        MPI_Irecv(&RecvBuf, SendProc[i], &RecvReq);
        UnpackReverse(RecvBuf, SendNum[i], SendList[i]);
    } else {
        SignalWait(&gData[SendNodeRank].RevSignal[i], 1);
        UnpackReverse(gData[SendNodeRank].Buffer,
                      SendNum[i], SendList[i]);
        SignalClear(&gData[SendNodeRank].RevSignal[i]);
    }

    if(RecvNodeRank == MPI_UNDEFINED)
        MPI_Wait(&SendReq);
    else
        SignalWait(&gData[NodeRank].FwdSignal[i], 0);
}
```

FIGURE A.7. MiniMD reverse communication phases transformed to use global (node-wide) locks and variables.

Forward Phase

```
for(i = 0; i < nSwap; i++) {
    HMPI_Comm_node_rank(RecvProc[i], &RecvNodeRank);

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Irecv(&RecvBuf, RecvProc[i], &RecvReq);
    } else {
        gData[NodeRank].Buffer = atom.x;
        gData[NodeRank].First = FirstRecv[i];
        SignalSet(&gData[NodeRank].FwdSignal[i]);
    }

    HMPI_Comm_node_rank(SendProc[i], &SendNodeRank);

    if(SendNodeRank == MPI_UNDEFINED) {
        PackForward(SendBuf, SendNum[i], SendList[i]);
        MPI_Send(&SendBuf, SendProc[i]);
    } else {
        SignalWait(&gData[SendNodeRank].FwdSignal[i], 1);
        ForwardFusion(gData[SendNodeRank].Buffer,
                      SendNum[i], SendList[i],
                      gData[SendNodeRank].First);
        SignalClear(&gData[SendNodeRank].FwdSignal[i]);
    }

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Wait(&RecvReq);
        UnpackForward(RecvBuf, RecvNum[i], FirstRecv[i]);
    } else {
        SignalWait(&gData[NodeRank].FwdSignal[i], 0);
    }
}
```

FIGURE A.8. MiniMD forward communication phase transformed to use loop fusion.

Reverse Phase

```
for(i = 0; i < nSwap; i++) {
    HMPI_Comm_node_rank(RecvProc[i], &RecvNodeRank);

    if(RecvNodeRank == MPI_UNDEFINED) {
        PackReverse(SendBuf, RecvNum[i], FirstRecv[i]);
        MPI_Isend(SendBuf, RecvProc[i], &SendReq);
    } else {
        gData[NodeRank].Buffer = atom.f;
        gData[NodeRank].First = FirstRecv[i];
        SignalSet(&gData[NodeRank].RevSignal[i]);
    }

    HMPI_Comm_node_rank(SendProc[i], &SendNodeRank);

    if(SendNodeRank == MPI_UNDEFINED) {
        MPI_Recv(RecvBuf, SendProc[i], &RecvReq);
        UnpackReverse(RecvBuf, SendNum[i], SendList[i]);
    } else {
        SignalWait(&gData[SendNodeRank].RevSignal[i], 1);
        ForwardFusion(gData[SendNodeRank].Buffer,
                      SendNum[i], SendList[i],
                      gData[SendNodeRank].First);
        SignalClear(&gData[SendNodeRank].RevSignal[i]);
    }

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Wait(&SendReq);
    } else {
        SignalWait(&gData[NodeRank].RevSignal[i], 0);
    }
}
```

FIGURE A.9. MiniMD reverse communication phase transformed to use loop fusion.

B

FFT2D Code Examples

In this appendix, we include listings for various forms of the FFT2D communication code. The communication is an all-to-all pattern executed twice per FFT operation. Both communication phases are identical. We show versions of the communication code with with MPI/HMPI (Chapter 3), OPI (Chapter 4), and the shared memory techniques discussed in Chapter 5.

```
int BlockSize = ProblemSize / NumRanks;

// Pack & Send loop
for(rank = 0; rank < NumRanks; rank++) {
    SrcBuf = &FFTBuf[rank * BlockSize];
    DestBuf = &SendBuf[rank * BlockSize * BlockSize];

    PackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);

    MPI_Isend(DestBuf, rank, SendReq[rank]);
}

for(rank = 0; rank < NumRanks; rank++)
    MPI_Irecv(&RecvBuf[rank * BlockSize * BlockSize],
              rank, RecvReq[rank]);

MPI_Waitall(NumRanks, SendReq);

// Receive & Unpack loop
for(rank = 0; rank < NumRanks; rank++) {
    MPI_Wait(&RecvReq[rank]);
    SrcBuf = &RecvBuf[rank * BlockSize * BlockSize];
    DestBuf = &FFTBuf[rank * BlockSize];

    UnpackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);
}
```

FIGURE B.1. MPI version of FFT2D all-to-all communication, forming a distributed matrix transposition.

```
int BlockSize = ProblemSize / NumRanks;

// Pack & Send loop
for(rank = 0; rank < NumRanks; rank++) {
    SrcBuf = &FFTBuf[rank * BlockSize];
    OPI_Alloc(&DestBuf, BlockSize * BlockSize);

    PackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);

    OPI_Igive(&DestBuf, rank, SendReq[rank]);
}

// Receive & Unpack loop
for(rank = 0; rank < NumRanks; rank++) {
    OPI_Take(&SrcBuf, rank);

    DestBuf = &FFTBuf[rank * BlockSize];

    UnpackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);
    OPI_Free(&SrcBuf);
}

MPI_Waitall(NumRanks, SendReq);
```

FIGURE B.2. OPI version of FFT2D all-to-all communication, forming a distributed matrix transposition.

```
void PackBlock(complex_t* DestBuf, complex_t* SrcBuf,
               int BlockSize, int ProblemSize)
{
    for(x = 0; x < BlockSize; x++) {
        for(y = 0; y < BlockSize; y++) {
            DestBuf[x * BlockSize + y] =
                    SrcBuf[x * Problemsize + y];
        }
    }
}

void UnpackBlock(complex_t* DestBuf, complex_t* SrcBuf,
                 int BlockSize, int ProblemSize)
{
    // Transpose while unpacking
    for(x = 0; x < BlockSize; x++) {
        for(y = 0; y < BlockSize; y++) {
            DestBuf[y * ProblemSize + x] =
                    SrcBuf[x * BlockSize + y];
        }
    }
}
```

FIGURE B.3. Original FFT2D pack and unpack (with transpose) routines.

```
typedef struct
{
    void* Buffer;

    counter_t Counter;
} GlobalData;

GlobalData* gData;
```

FIGURE B.4. Node-wide data structure for shared locks and variables in FFT2D.

```
int BlockSize = ProblemSize / NumRanks;

for(rank = 0; rank < NumRanks; rank++) {
    SrcBuf = &FFTBuf[rank * BlockSize];
    DestBuf = &SendBuf[rank * BlockSize * BlockSize];

    PackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);

    HMPI_Comm_node_rank(rank, &SendNodeRank);

    if(SendNodeRank == MPI_UNDEFINED)
        MPI_Isend(DestBuf, rank, SendReq[rank]);
}

gData[NodeRank].Buffer = SendBuf;
Counter = CounterAdd(&gData[NodeRank].Counter, 1);

for(rank = 0; rank < NumRanks; rank++)
    MPI_Irecv(&RecvBuf[rank * BlockSize * BlockSize],
              rank, RecvReq[rank]);

MPI_Waitall(NumRanks, SendReq);

for(rank = 0; rank < NumRanks; rank++) {
    HMPI_Comm_node_rank(rank, &RecvNodeRank);

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Wait(&RecvReq[rank]);

        SrcBuf = &RecvBuf[rank * BlockSize * BlockSize];
    } else {
        CounterWaitGT(&gData[RecvNodeRank].Counter, Counter);

        int BufferIndex = rank * BlockSize * BlockSize;
        SrcBuf = gData[RecvNodeRank].Buffer[BufferIndex];
    }

    DestBuf = &FFTBuf[rank * BlockSize];
    UnpackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);
}

Counter = CounterAdd(&gData[NodeRank].Counter, 1);

for(t = 0; t < NodeSize; t++)
    CounterWaitGT(&gData[RecvNodeRank].Counter, Counter);
```

FIGURE B.5. FFT2D all-to-all communication transformed for shared memory synchronization.

```
int BlockSize = ProblemSize / NumRanks;

gData[NodeRank].Buffer = FFTBuf;
Counter = CounterAdd(&gData[NodeRank].Counter, 1);

for(rank = 0; rank < NumRanks; rank++) {
    HMPI_Comm_node_rank(rank, &SendNodeRank);

    if(SendNodeRank == MPI_UNDEFINED) {
        SrcBuf = &FFTBuf[rank * BlockSize];
        DestBuf = &SendBuf[rank * BlockSize * BlockSize];

        PackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);
        MPI_Isend(DestBuf, rank, SendReq[rank]);
    } else {
        CounterWaitGT(&gData[SendNodeRank].Counter, Counter);

        SrcBuf = &gData[SendNodeRank].Buffer[MyRank * BlockSize;
        DestBuf = &FFTBuf[rank * BlockSize];

        FuseBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);
    }
}

Counter = CounterAdd(&gData[NodeRank].Counter, 1);

for(t = 0; t < NodeSize; t++)
    CounterWaitGT(&gData[RecvNodeRank].Counter, Counter);

for(rank = 0; rank < NumRanks; rank++)
    MPI_Irecv(&RecvBuf[rank * BlockSize * BlockSize],
              rank, RecvReq[rank]);

MPI_Waitall(NumRanks, SendReq);

for(rank = 0; rank < NumRanks; rank++) {
    HMPI_Comm_node_rank(rank, &RecvNodeRank);

    if(RecvNodeRank == MPI_UNDEFINED) {
        MPI_Wait(&RecvReq[rank]);

        DestBuf = &FFTBuf[rank * BlockSize];
        SrcBuf = &RecvBuf[rank * BlockSize * BlockSize];

        UnpackBlock(DestBuf, SrcBuf, BlockSize, ProblemSize);
    }
}
```

FIGURE B.6. FFT2D all-to-all communication transformed for loop fusion.

```
void FuseBlock(complex_t* DestBuf, complex_t* SrcBuf,
               int BlockSize, int ProblemSize)
{
    for(x = 0; x < BlockSize; x++) {
        for(y = 0; y < BlockSize; y++) {
            DestBuf[y * ProblemSize + x] =
                    SrcBuf[x * ProblemSize + y];
        }
    }
}
```

FIGURE B.7. FFT2D loop fusion routine.

C

Hybrid MPI Source Code

Alongside this dissertation we have developed the HMPI library, which contains code implementing the shared memory heap allocator, the *Immediate* and *Synergistic* data transfer protocols, and the Ownership Passing Interface OPI. HMPI has been made available as open source software at the following location:

`https://code.google.com/p/hmpi/`

We hope that the community will find this useful for applying our ideas to applications, integrating into existing MPI libraries, and/or for continuing related research.

Bibliography

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Jr, and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007.

[2] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. 1999.

[3] Dan Bonachea. Gasnet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[4] Shekhar Borkar. Will interconnect help or limit the future of computing. In *Conference on Hot Interconnects*, 2011.

[5] Ron Brightwell. Exploiting direct access shared memory for mpi on multi-core processors. *International Journal of High Performance Computing Applications*, 24:69–77, February 2010.

[6] Ron Brightwell, Kevin Pedretti, and Trammell Hudson. Smartmap: operating system support for efficient data sharing among processes on a multi-core processor. In *Supercomputing*, pages 25:1–25:12, 2008.

[7] Greg Bronevetsky, Andrew Friedley, Torsten Hoefler, Andrew Lumsdaine, and Dan Quinlan. Compiling mpi for many-core systems. Technical Report LLNL-TR-638557, Lawrence Livermore National Laboratory, 2013.

[8] Darius Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud. Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis. In *International Conference on Parallel Processing*. IEEE Computer Society, April 2009.

[9] Darius Buntinas, Guillaume Mercier, and William Gropp. Implementation and evaluation of shared-memory communication and synchronization operations in mpich2 using the nemesis communication subsystem. *Journal of Parallel Computing*, 33(9), September 2007.

[10] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley, Boston, MA, USA, 1997.

[11] David Callahan. A global approach to detection of parallelism. Technical report, Rice University, 1987.

[12] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The cascade high productivity language. In *International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 52–60, 2004.

[13] Lei Chai, Albert Hartono, and Dhabaleswar K. Panda. Designing high performance and scalable mpi intra-node communication support for clusters. In *International Conference on Cluster Computing*, 2006.

[14] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Conference on Object-oriented programming, systems, languages, and applications*, pages 519–538. ACM, 2005.

[15] Eleanor Chu and Alan George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. 1999.

[16] GNU Compiler Collection. Atomic builtins. `http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html`.

[17] Intel Corporation. Intel itanium processor-specific application binary interface (ABI), May 2001.

[18] Intel Corporation. Intel 64 and ia-32 architectures software developer manuals, June 2013.

[19] A. Danalis, L. Pollock, M. Swany, and J. Cavazos. MPI-aware compiler optimizations for improving communication-computation overlap. In *International Conference on Supercomputing*, June 2009.

[20] Erik Demaine. A threads-only mpi implementation for the development of parallel programs. In *International Symposium on High Performance Computing Systems*, pages 153–163, 1997.

[21] Yuri Dotsenko. Expressiveness, programmability and portable high performance of global address space languages. Technical report, Rice University, 2006.

[22] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Memory-Access Times. *Electronics*, 57(1), January 1984.

[23] Andrew Friedley, Greg Bronevetsky, Torsten Hoefler, and Andrew Lumsdaine. Hybrid MPI: Efficient message passing for multi-core systems. In *Supercomputing*, November 2013.

[24] Andrew Friedley, Torsten Hoefler, Greg Bronevetsky, Ching-Chen Ma, and Andrew Lumsdaine. Ownership passing: Efficient distributed memory programming on multi-core systems. February 2013. Symposium on Principles and Practice of Parallel Programming.

[25] Andrew Friedley and Andrew Lumsdaine. Communication optimization beyond MPI. May 2011. International Parallel and Distributed Processing Symposium.

[26] 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 *European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[27] Edgar Gabriel, Michael Resch, and Roland Rhle. Implementing mpi with optimized algorithms for metacomputing, 1999.

[28] Brice Goglin and Stéphanie Moreaud. KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. *Journal of Parallel and Distributed Computing*, 73(2), February 2013.

[29] X Gonze, G Rignanese, M Verstraete, J Betiken, Y Pouillon, R Caracas, F Jollet, M Torrent, G Zerah, M Mikami, and et al. A brief introduction to the ABINIT software package. *Zeitschrift fr Kristallographie*, 220(5-6-2005), 2005.

[30] Andrew Grover and Christopher Leech. Accelerating network receive processing (Intel I/O acceleration technology). In *Linux Symposium*, pages 281–288, Ottawa, Canada, July 2005.

[31] Michael A. Heroux, Douglas W. Dorfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Rober W. Numrich. Improving performance via mini-applications. 2009.

[32] Paul N. Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L. Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, and Katherine A. Yelick. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, EECS Department, University of California, Berkeley, Nov 2005.

[33] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur. Leveraging MPI's One-Sided Communication Interface for Shared-Memory Programming. In *European MPI Users' Group Meeting*, Sep. 2012.

[34] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur. MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Journal of Computing*, May 2013.

[35] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread? In *IEEE International Conference on Cluster Computing*, October 2008.

[36] Wilson C. Hsieh and William E. Weihl. Scalable reader-writer locks for parallel systems. In *International Parallel Processing Symposium*, pages 656–659, 1992.

[37] Chao Huang, Orion Lawlor, and Laxmikant V. Kale. Adaptive MPI. In *International Workshop on Languages and Compilers for Parallel Computing*, College Station, Texas, October 2003.

[38] Inc. IBM. A2 processor user's manual for Blue Gene/Q, October 23 2012.

[39] Inc. IBM. IBM XL C/C++ for Blue Gene, V12.1 Compiler Reference, 2012.

[40] Google Inc. gperftools. `https://code.google.com/p/gperftools`.

[41] Jian Ke and Evan Speight. Tern: Thread migration in an mpi runtime environment. Technical Report CSL-TR-2001-1016, Cornell Computer Systems Laboratory, November 2001.

[42] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still.

Exploring traditional and emerging parallel programming models using a proxy application. In *International Parallel and Distributed Processing Symposium*, May 2013.

[43] Rainer Keller, Edgar Gabriel, Bettina Krammer, Matthias S. Mller, and Michael M. Resch. Towards efficient execution of mpi applications on the grid: Porting and optimization issues. *Journal of Grid Computing*, 2003.

[44] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320. Springer-Verlag, 1994.

[45] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[46] Orran Krieger, Michael Stumm, Ron Unrau, and Jonathan Hanna. A fair fast scalable reader-writer lock. In *International Conference on Parallel Processing*, pages 201–204. CRC Press, 1993.

[47] S. Kumar, C. Huang, G. Zheng, E. Bohm, A. Bhatele, J. C. Phillips, H. Yu, and L. V. Kalé. Scalable molecular dynamics with NAMD on the IBM Blue Gene/L system. *IBM J. Res. Dev.*, 52, January 2008.

[48] Sandia National Laboratories. Large-scale atomic/molecular massively parallel simulator (LAMMPS). `http://lammps.sandia.gov`.

[49] Sandia National Laboratories. Mantevo. `http://mantevo.org`.

[50] Lawrence Livermore National Laboratory. Simulation of underwater explosion benchmark experiments with ale3d. Technical Report UCRL-JC-123819, May 1997.

[51] Lawrence Livermore National Laboratory. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254, 2010.

[52] Doug Lea. Doug Lea's malloc (dlmalloc). `http://g.oswego.edu/dl/html/malloc.html`.

[53] Lie-Quan Lee and Andrew Lumsdaine. Generic programming for high performance scientific applications. In *Proceedings of the 2002 Joint ACM Java Grande – ISCOPE Conference*, 2002.

[54] Lie-Quan Lee and Andrew Lumsdaine. The generic message passing framework. In *International Parallel and Distributed Processing Symposium*, April 2003.

[55] Yossi Lev, Victor Luchangco, and Marek Olszewski. Scalable reader-writer locks. In *Symposium on Parallelism in algorithms and architectures*, pages 101–110, 2009.

[56] S. Li, T. Hoefler, , and M. Snir. NUMA-Aware Shared Memory Collective Communication for MPI. In *International Symposium on High Performance Distributed Computing*, 2013.

[57] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. High performance rdma-based mpi implementation over infiniband. In *International Conference on Supercomputing*, 2003.

[58] Zhiqiang Liu, Kaijun Ren, and Junqiang Song. Mpiactor - a multicore-architecture adaptive and thread-based mpi program accelerator. In *International Conference on High Performance Computing and Communications*, pages 98–107, Washington, DC, USA, 2010. IEEE Computer Society.

[59] Teng Ma, George Bosilca, Aurlien Bouteiller, Brice Goglin, Jeffrey M. Squyres, and Jack J. Dongarra. Kernel Assisted Collective Intra-node Communication Among Multicore and Manycore CPUs. Technical report, INRIA, December 2010.

[60] Teng Ma, Aurelien Bouteiller, George Bosilca, and Jack J. Dongarra. Impact of kernel-assisted mpi communication over scientific applications: Cpmd and fftw. In *European MPI Users' Group Meeting*, 2011.

[61] Nilesh Mahajan, Uday Pitambare, and Arun Chauhan. Globalizing selectively: Shared-memory efficiency with address-space separation. Technical report, November 2013.

[62] Megan Gilge. IBM system Blue Gene solution: Blue Gene/Q application development, December 20 2012.

[63] John M. Mellor-crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9, 1991.

[64] John M. Mellor-crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Symposium on Principles and practice of parallel programming*, pages 106–113, 1991.

[65] MPI Forum. MPI: A message-passing interface standard. version 2.2, September 4 2009.

[66] MPI Forum. MPI: A message-passing interface standard. version 3, September 21 2012.

[67] Aaftab Munshi. The OpenCL specification version 1.0, 2009.

[68] Stas Negara, Rajesh K. Karmani, and Gul Agha. Inferring ownership transfer for efficient message passing. In *Symposium on Principles and practice of parallel programming*, New York, NY, USA, 2011.

[69] Stas Negara, Gengbin Zheng, Kuo-Chuan Pan, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kale, and Paul M. Ricker. Automatic MPI to AMPI Program Transformation using Photran. In *Workshop on Productivity and Performance*, Ischia, Naples, Italy, August 2010.

[70] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *ACM FORTRAN FORUM*, 17(2):1–31, 1998.

[71] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[72] Kevin Pedretti and Brian Barrett. XPMEM: Cross-Process Memory Mapping.

[73] S. Pellegrini, T. Hoefler, and T. Fahringer. On the Effects of CPU Caches on MPI Point-to-Point Communications. In *International Conference on Cluster Computing*, 2012.

[74] Marc Pérache, Patrick Carribault, and Hervé Jourdren. MPC-MPI: An MPI implementation reducing the overall memory consumption. In *European PVM/MPI Users' Group Meeting*, Berlin, Heidelberg, 2009.

[75] Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *JOURNAL OF COMPUTATIONAL PHYSICS*, 117:1–19, 1995.

[76] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. 1992.

[77] Jelica Protic, Milo Tomasevic, and Veljko Milutinovic, editors. *Distributed Shared Memory: Concepts and Systems*. Los Alamitos, CA, USA, 1st edition, 1997.

[78] Daniel J. Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[79] Daniel J. Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik. Parallel object-oriented framework optimization: Research articles. *Journal of Concurrent Computing : Practical Experience*, 16(2-3):293–302, January 2004.

[80] Rolf Rabenseifner. Hybrid parallel programming on HPC platforms. In *European Workshop on OpenMP*, Aachen, Germany, 2003.

[81] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *International Conference on Parallel, Distributed and Network-based Processing*, Washington, DC, USA, 2009.

[82] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, USA, 4th edition, 2013.

[83] H. Tang and T. Yang. Optimizing threaded MPI execution on SMP clusters. In *International Conference on Supercomputing*, 2001.

[84] Rajeev Thakur and William Gropp. Test suite for evaluating performance of multithreaded MPI communication. *Journal of Parallel Computing*, 35(12), December 2009.

[85] Dave Turner. Optimizing SMP message-passing systems. `http://drdaveturner.com/pubs/ANL_04.ppt`, 2004.

[86] Dave Turner and Xuehua Chen. Protocol-dependent message-passing performance on linux clusters. In *International Conference on Cluster Computing*, Washington, DC, USA, 2002.

[87] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[88] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The SGI Altix 3000 global shared-memory architecture. 2005.

[89] Hyun wook Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K. Panda. LIMIC: Support for high-performance mpi intra-node communication on linux cluster. In *International Conference on Parallel Processing*, pages 184–191, 2005.

[90] Shin yuan Tzou, David, and David P. Anderson. The performance of message-passing using restricted virtual memory remapping. *Journal of Software - Practice and Experience*, 21, 1991.