Hybrid MPI: Efficient Message Passing for Multi-core Systems

Andrew Friedley
Indiana University

Greg Bronevetsky
Lawrence Livermore National Laboratory

Torsten Hoefler
ETH Zurich

Andrew Lumsdaine
Indiana University

ABSTRACT

Multi-core shared memory architectures are ubiquitous in both High-Performance Computing (HPC) and commodity systems because they provide an excellent trade-off between performance and programmability. MPI's abstraction of explicit communication across distributed memory is very popular for programming scientific applications. Unfortunately, OS-level process separations force MPI to perform unnecessary copying of messages within shared memory nodes. This paper presents a novel approach that transparently shares memory across MPI processes executing on the same node, allowing them to communicate like threaded applications. While prior work explored thread-based MPI libraries, we demonstrate that this approach is impractical and performs poorly in practice. We instead propose a novel process-based approach that enables shared memory communication and integrates with existing MPI libraries and applications without modifications. Our protocols for shared memory message passing exhibit better performance and reduced cache footprint. Communication speedups of more than 26% are demonstrated for two applications.

1. INTRODUCTION

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 MPI implementations provide for many-core shared memory architectures has forced developers to use alternative programming models such as OpenMP [17] or OpenCL [1] to parallelize computations on such hardware, using MPI only for inter-node communication.

Despite the limitations of today's MPI implementations, its programming model is actually highly compatible with the needs of future applications. In particular, since communication is expected to make up the bulk of application power use in the future, algorithms are expected to limit their communication and memory use. MPI simplifies this by defaulting memory to be private to each computation thread and requiring the developer to explicitly indicate any communication. The challenge of MPI implementations is to provide developers this efficient abstraction across a wide range of architectures, including those where memory is actually shared or where only restricted communication primitives are available.

This paper 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 MPI (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. The resulting composition of HMPI for intra-node communication with a traditional MPI for inter-node communication produces a comprehensive communication system for clusters of shared-memory nodes. We demonstrate this experimentally by composing HMPI with MVAPICH2 [13] and Blue Gene/Q MPI [14].

The intuition of our design is that efficient use of shared memory hardware requires the memories of MPI ranks running on the same shared memory 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). As discussed in Section 2 this makes it possible for MPI to transfer data directly from a sender's buffer to a receiver's buffer with no additional copies to overcome OS separations between their address spaces. Further, it enables novel shared-memory optimizations such as ownership passing [2] where the sender passes a pointer to its data buffer to the receiver, allowing the receiver to copy the data directly into its internal data structures without the need to first copy it into a receive buffer. 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.

These observations have also been made by the MPI Forum, which introduced shared memory windows in MPI-3.0 [6]. 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. We will demonstrate how much of the benefits of shared memory communication can be utilized without changing applications.

Prior work on sharing memory across MPI ranks has focused on implementing ranks as threads within the same OS process. While multiple threads appear to be the natural solution to sharing memory, this approach suffers from several challenges that reduce its generality and performance. In our approach, we transparently share heap memory among OS processes. This transition from threads to processes enables HMPI to work seamlessly with existing MPI applications and without the performance issues of the thread-oriented approach. Our approach works entirely in user space on commodity x86 systems with no kernel extensions or modification of system libraries.

Sections 4, 5, and 6 experimentally evaluate the performance of our approach, showing that it outperforms native MPI libraries on multiple benchmarks. We analyze HMPI's affect on the processor cache in detail to show that it also improves performance by utilizing caches more efficiently and interfering less with the application's own use of the cache.

The key contributions of our work are the following:

1. An analysis of thread-based MPI design and identification of its limitations.
2. A shared memory allocator technique for transparently enabling shared memory between local MPI ranks without modifying application code.
3. Two new point-to-point protocols for message passing that utilize a shared address space for better performance.
4. Analysis of shared-memory message passing performance on an x86 system using our shared memory allocator technique, and on Blue Gene/Q, a system providing a shared address space feature.

2. MPI ON SHARED MEMORY SYSTEMS

In this paper, we discuss three different approaches to memory layout and intra-node communication in MPI. Section 2.1 discusses the traditional (process-based) approach used in practice by most implementations. 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. For reasons we will discuss in Section 2.2, this approach is not prevalent in practice. We contribute a third approach, discussed in Sections 3 and 4, that assigns each rank to its own process but shares heap memory among all ranks in a node. Our approach combines the benefits of process-based and thread-based MPI design.

2.1 Process-based MPI

Although the MPI standard 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 1 illustrates this layout. The advantage of the process-based design is that it makes it easier 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. Since network interfaces are typically designed to provide each process a separate context in which to coordinate its outgoing and incoming communication, no MPI-level synchronization is required to access the network.

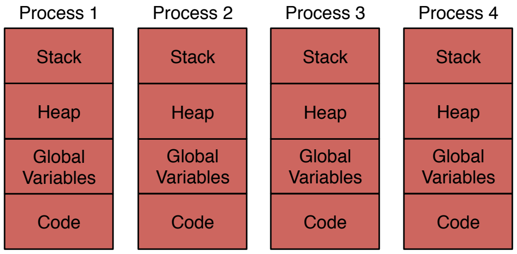

Figure 1: Memory layout in the traditional process-based MPI design. No application-visible memory is shared when MPI ranks are processes.

The limitation of this design is in communicating among different ranks that are executing within the same shared memory node. MPI libraries often use a FIFO connection (one for each pair of local ranks) for small messages, and one or more shared memory regions mapped by multiple ranks for larger messages. Either case inherently requires two copy operations per message. The sender copies from its private-memory send buffer into the FIFO queue or shared memory region, and the receiver copies back out into its separate private-memory receive buffer. 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.

FIFOs are a pair-wise connection, and shared memory regions may also be created on a pair-wise basis to simplify communication. Thus, the number of resources grows as the square of the number of MPI ranks per node, which is often the number of cores per node. Such an approach will consume too many resources as the amount of memory available per core continues to decrease on HPC systems.

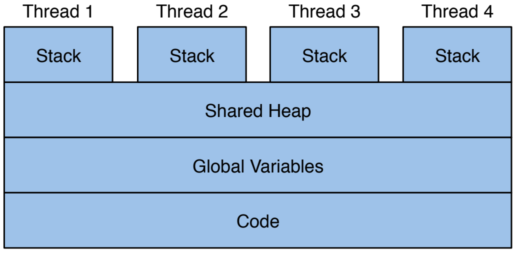

Figure 2: Memory layout in the thread-based MPI design. All application visible memory is shared when MPI ranks are threads.

2.2 Thread-based MPI

The limitations 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 [7, 20, 21]. Figure 2 illustrates this layout. Threads are an excellent choice since they share all their memory by default. 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 he 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 [16, 20]. There has been work on tools that modify object files to privatize global variables when the source code is not available [16].

Given the complexity of privatization, especially in library code, an alternative approach is to adjust the use of libraries to ensure that no globals 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., 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.

2.2.1 *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. For example, 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 3 shows the effect of this contention on the MiniMD[1] application. We measured the time taken by the communication portions of MiniMD, using varying numbers of ranks on 16 nodes of the Cab system. Cab has an InfiniBand-based Performance Scaled Messaging (PSM) network that is not thread-safe. Network resource contention will be a problem when using any network that does not support multiple threads as efficiently as multiple processes.

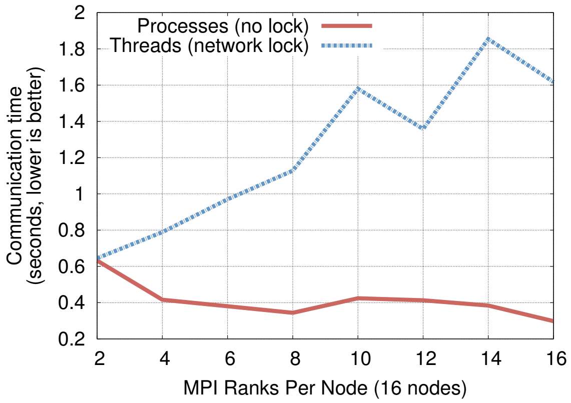

Figure 3: Affect of overhead due to lock contention for network resources in the MiniMD application with strong scaling. However, when using threads, the communication overhead increases due to lock contention.

Driver thread-safety also creates issues when using multiple threads with a process-based MPI implementation. If MPI is running in MPI_THREAD_SINGLE mode, application threads must use locking to control access to MPI, which creates contention. However, if MPI provides MPI_THREAD_MULTIPLE 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 one of the reasons why multi-threaded MPI applications are not common and perform sub-optimally [22].

All in all, the above observations suggest that thread-based MPI implementations are not a practical approach for the future of multi- and many-core HPC systems. We note that no thread-based MPI has been widely adopted in practice.

3. 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. The challenge of this approach is to share memory across processes by developing some mechanism to circumvent the memory protections typically enforced by the OS.

A number of solutions for sharing memory between processes already exist on HPC systems. XPMEM [18, 24] is a Linux kernel extension that allows processes to map memory from another process, and is commonly found on SGI and Cray systems. Blue Gene/Q systems have a fea-

[1] See Section 6.1 for details on MiniMD.

ture that shares heap memory among all MPI ranks in a node, and is enabled by setting an environment variable (BG_MAPCOMMONHEAP) [14]. LIMIC [25] and KNEM [4] are kernel extensions adopted by MVAPICH and Open MPI, respectively, that enable single copy communication. However, they require modification of the OS kernel, and due to the overhead of system calls only provide performance benefits for large messages. The LIMIC authors report in [25] that there is only an improvement for 4 KiB and larger message sizes. [4] states that KNEM is not competitive for small messags, usually less than 16 KiB.

For Linux-based clusters without such 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 all 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.

3.1 Shared Memory Heap Allocator

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. Normally, the memory allocator incrementally requests memory from the operating system using the `sbrk` or `mmap` system calls. We implement our own version of `sbrk` that requests memory from a shared memory region mapped on all MPI processes. Using this approach, we have modified both Doug Lea's malloc library (dlmalloc) [11] and Google's tcmalloc [8] library to transparently provide shared memory from `malloc` and related routines. All HMPI results shown in this paper use our modified dlmalloc library, while the MPI results use the Linux default system allocator (which is also based on dlmalloc).

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 using `mmap`. The shared region is divided evenly among the ranks on the node, and each rank allocates memory only from its part of the region. This approach eliminates the need for any synchronization between processes within the memory allocator. On the Cab system with 32 GiB RAM per node, we were able to reserve 32 GiB of virtual memory per MPI rank, for a total of $32GiB \cdot 16ranks = 512GiB$. Without swap, total memory usage cannot exceed physical memory, but this larger shared region allows for unbalanced memory usage across ranks.

Figure 4 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.

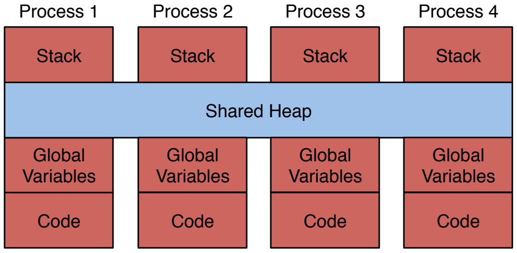

Figure 4: Memory layout of processes with our shared heap allocator. Dark red segments are private to each process, while the light blue heap segment is shared among all processes in a node.

In addition to the heap, MPI allows communication buffers located in global variables and the stack segment. Section 2.2 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. Our library falls back to single-copy communication if the receiver's buffer is not shared, and two-copy communication if the sender's buffer is not shared (see Section 4.2).

4. HYBRID MPI

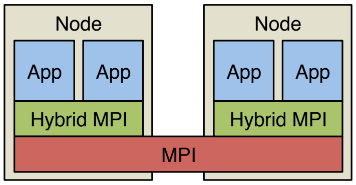

Figure 5: Hybrid MPI sits between applications and an existing MPI. Intra-node communication is handled by HMPI, while an MPI is used across nodes.

We have implemented a 'Hybrid' MPI (HMPI) library to investigate single-copy message passing techniques. 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 5 illustrates how HMPI layers between applications and MPI. We did not use the PMPI (Profiling MPI) interface, but instead redefined MPI routines using a header file. PMPI remains instact at the underlying MPI layer below HMPI, though we believe it can be supported at the HMPI layer as well. 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 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. In this paper, we focus on point-to-point techniques in HMPI. Prior work [12] has explored NUMA-aware collective communication algorithms using HMPI and shared memory.

PACX-MPI [3, 9] 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.

To implement shared-memory message passing, HMPI assumes that memory used for communication is mapped to the same virtual address in every process on a node. We show experimental results using our shared memory heap allocator on a commodity x86 cluster ('Cab'), and using the BG_MAPCOMMONHEAP feature on Blue Gene/Q ('Sequoia'). Section 6 describes these systems in detail.

4.1 Message Matching

We implement two incoming message queues per receiver using linked lists. Figure 6 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 an MCS lock [15]. An important benefit of the MCS lock is guaranteed 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.

The second queue is private. When a receiver attempts to match incoming sends to local receives, it drains its global queue and adds incoming sends to its private queue. Since the queues are linked lists, the draining operation only involves updating two 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. Our dual queue technique minimizes contention between processes.

4.2 Communication Protocols

Although single-copy message transfer was our goal with HMPI, we have discovered that simply using `memcpy` to transfer the data is often not the fastest method possible. We use an 'immediate' protocol for small messages less than the *immediate threshold* (currently 256 bytes, Section 4.2.1), and a 'synergistic' protocol for messages larger than the *synergistic threshold* (currently 4 or 16 kB, Section 4.2.2). 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.

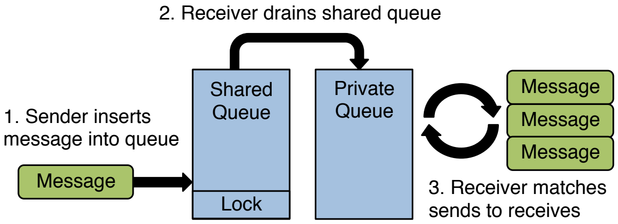
2. Receiver drains shared queue
1. Sender inserts message into queue
Message
Shared Queue
Lock
Private Queue
Message
Message
Message
3. Receiver matches sends to receives

Figure 6: 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.

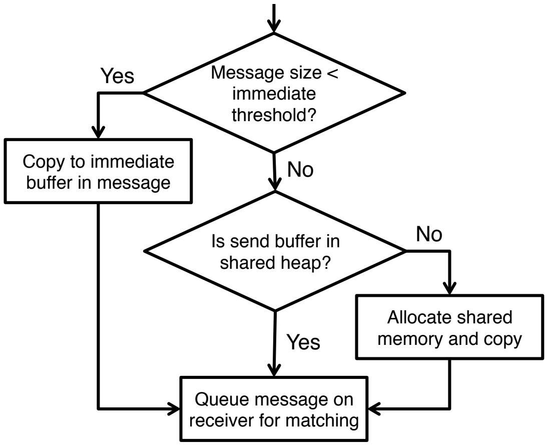
Message size < immediate threshold?
Yes
Copy to immediate buffer in message
No
Is send buffer in shared heap?
No
Allocate shared memory and copy
Yes
Queue message on receiver for matching

Figure 7: Sender protocol flow. The sender ensures its buffer is in the shared heap and uses the immediate transfer protocol for small messages.

Before queuing a message, the sender goes through a series of checks as shown in Figure 7. If the message is small, we go into 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. Finally, the message is queued on the receiver's shared queue.

Once a message is matched, the receiver decides how to transfer data from the send buffer to the receive buffer. Figure 8 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 protocol, we use `memcpy` to transfer the data. For larger messages we enter the synergistic protocol.

4.2.1 Immediate Transfer Protocol

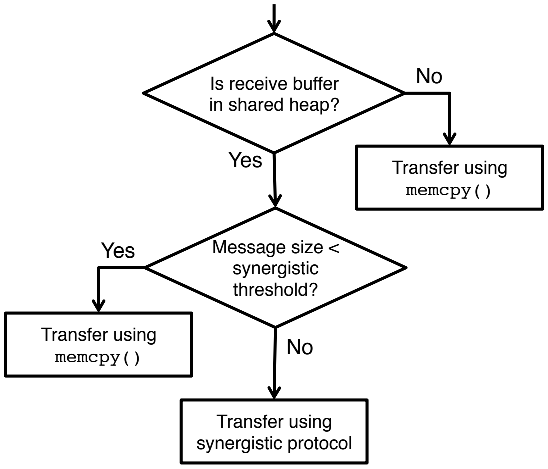
Is receive buffer in shared heap?
No
Transfer using memcpy()
Yes
Message size < synergistic threshold?
Yes
Transfer using memcpy()
No
Transfer using synergistic protocol

Figure 8: 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.

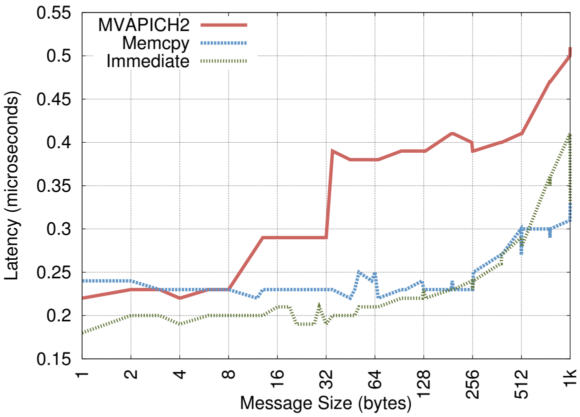

Figure 9: Intra-socket small message latency on Cab.

For small messages, the best latency is achieved by utilizing a two-copy method with the message data located immediately after the matching information. 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 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 7, 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. For small messages, the time saved by avoiding the cache miss more than makes up for the cost of doing two copies.

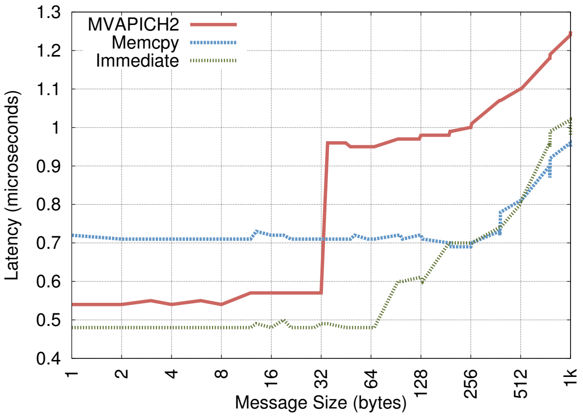

Figure 10: Inter-socket small message latency on Cab.

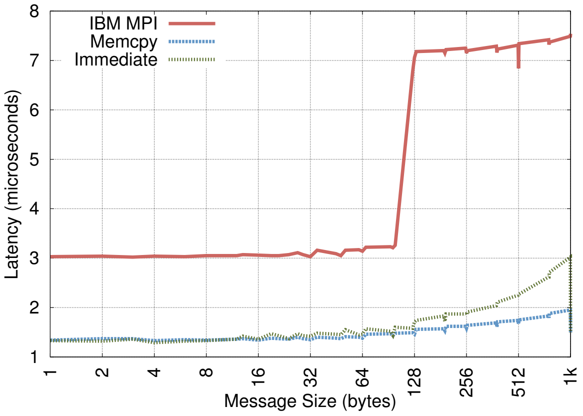

Figure 11: Small message latency on Sequoia.

Figures 9 and 10 show intra- and inter-socket small message latency on Cab[3] using the NetPIPE [23] benchmark. Based on these results, we chose a threshold of 256 bytes, below which we use the immediate protocol. Above that, we use `memcpy` or the synergistic protocol. Figure 11 shows small message latency on the Sequoia[3] system (one socket per node). Since we observe no benefit from the immediate protocol on this system, we conditionally disable it if compiling for a Blue Gene/Q machine and fall back to `memcpy`.

4.2.2 *Synergistic transfer protocol*

For large messages, bandwidth is most important. 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. 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 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

[3] Details on the Cab and Sequoia systems can be found in Section 6.

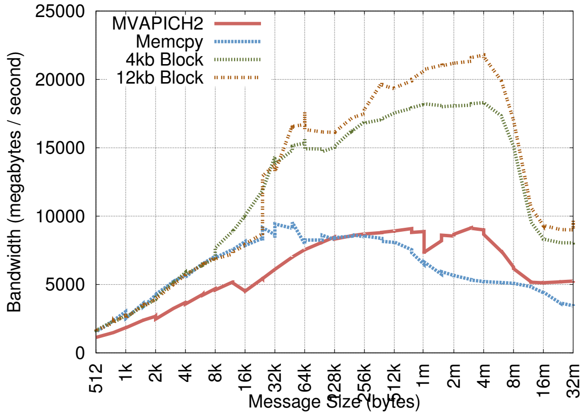

Figure 12: Intra-socket large message bandwidth on Cab.

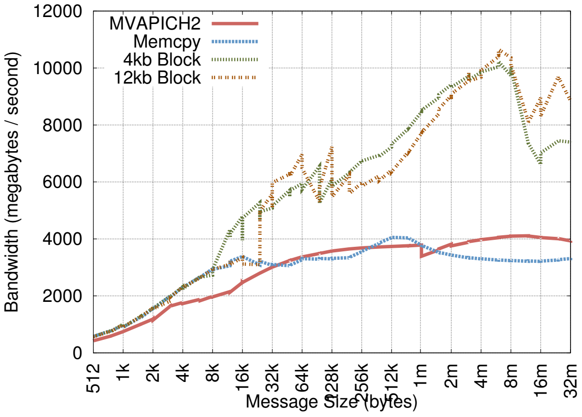

Figure 13: Inter-socket large message bandwidth on Cab.

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. Communication performance is dynamically accelerated when the sender is able to assist the receiver in copying data.

Figures 12 and 13 show intra- and inter-socket large message bandwidth on the Cab 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 14 shows large message bandwidth on Sequoia, which has one socket per node. Based on experimental results, we chose block sizes of 16kb and 64kb.

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.

5. COMMUNICATION ANALYSIS

While raw communication performance is important, another way that MPI libraries affect application performance is their effects on the cache and the application data structures within it [19]. We thus studied the effect that HMPI's and MPI's communication protocols have on the cache via the micro-benchmark in Figure 15, 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 Cab with buffer sizes between 128 bytes and 32kb (the size of Cab'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 ≤ 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.

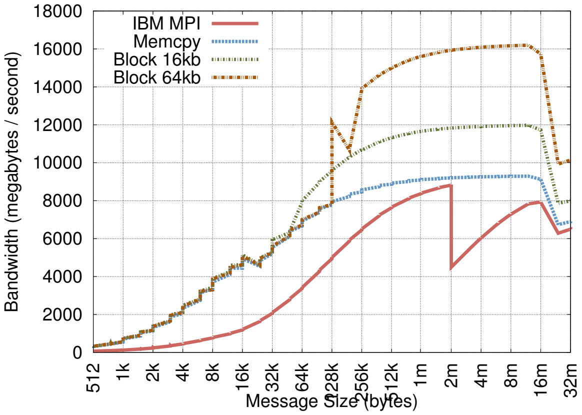

Figure 14: Large message bandwidth on Sequoia.

Figure 16 shows the misses in the L1 data cache on the Cab system (32kb in size), showing the average of 10 runs. Specifically, we report the fraction of the data buffer reads that miss: (number of L1 data misses) $*$ (cache line size)/(data buffer size). From top to bottom the graph shows misses for cases where HMPI is used for communication, then MPI and finally the case where no communication was performed between the read loops. From left to right we show data for the `sequential` and `random` loop orders and configurations where the buffers were the `common` or `separate`. For each configuration we show a heat map where the x-axis is communication

```
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 15: Benchmark that models the impact of MPI communication on application cache use.

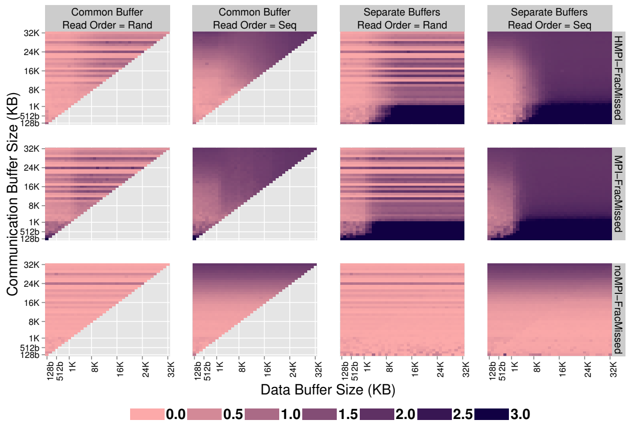

Figure 16: Fraction of accesses to the data buffer on which an L1 cache miss occurs. Lighter colors indicate fewer cache misses.

buffer size and y-axis is the data buffer size with the range $[128b, 256b, ...896b, 1kb, 2kb, ...32kb]$. The shade of each tile denotes the above fraction miss metric, with all values >=3 shown in the same shade to provide high visual resolution for the primary value range of [0-3].

The data shows that the miss rate is low when no communication is performed. It grows with the size of the data buffer up to the 32kb size of the L1 data cache. This phenomenon is observed for both read orders but is more significant for `sequential`. Since the `sequential` access pattern can be readily detected by the cache hardware, this indicates that the cache replacement algorithm on this processor is making sub-optimal replacement decisions.

Looking at the HMPI and MPI data, we observe both libraries increase the number of cache misses in the read loop. This is caused both by evictions of data from the cache as well as due to poor decisions by the cache replacement algorithm. The impact of cache evictions can be seen in the increased miss fractions for the `separate` buffer configuration relative to `common` buffer, since in the former case the processor touches more individual addresses.

The impact of the replacement algorithm is observed by looking at the difference between the results for `random` and `sequential` access orders. It can be seen that the difference between `separate` and `common` is generally small for all the buffer size configurations the two share. In contrast, the `sequential` access order causes many more misses than the `random`, indicating that the increase is due to interference with the replacement policy. The likely cause is that the cache access pattern of the communication code trains the replacement algorithm to expect the same access pattern in subsequent code and when control returns to the application it makes poor decisions that result in unnecessary misses. Indeed, in many cases there are more misses than the number of lines in the data buffer, especially for small data buffers and large communication buffers. This indicates that the useless lines from the communication buffer are being prefetched when in fact the application is attempting to access the data buffer.

Figure 17 shows the miss fraction metric of HMPI divided by the same metric of MPI. Values closer to 0 (HMPI has fewer misses) are shown in light shades while values close to 1.0 (HMPI and MPI are the same) are shown as dark. The data shows that for `random` reads HMPI induces fewer application misses across all data buffer sizes when communication buffers are smaller than 8kb (quarter of the L1 cache). The same is true for `sequential` reads where data buffers are smaller than 16kb (half the L1 cache) The conclusion is that 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 computing cores), HMPI has a significantly smaller impact on the application's use of the cache.

6. APPLICATION ANALYSIS

The various micro-benchmark results shown in Sections 4 and 5 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 applica-

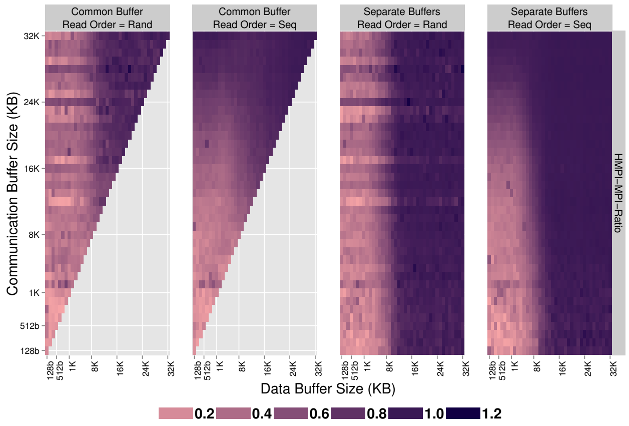

Figure 17: The fraction of accesses to the data buffer on which an L1 cache miss occurs with HMPI, divided by the same with MPI. Lighter colors indicate fewer misses in HMPI compared to MPI.

tions: 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.

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. Cab has two Xeon ES-2670 (eight core, 2.6 GHz) processors (16 cores total), 32 GiB of RAM per node, and a PSM InfiniBand network. MVAPICH2 v1.9rc1 was used as our comparison MPI on Cab. Sequoia is a Blue Gene/Q system with one PowerPC A2 (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 Sequoia.

6.1 MiniMD

MiniMD is part of the Mantevo [5] mini-application suite, which consists of several mini-applications representing larger application classes. Such mini-applications are increasingly used in exascale research for their combination of simplicity and relevance. MiniMD is a molecular dynamics simulation that computes atom movement over a 3D space decomposed into a processor grid. The primary communication pattern is a 3D, 6-point nearest neighbor exchange performed twice per work iteration.

Figures 18 and 19 show performance comparisons for MiniMD. We ran 2500 iterations and scaled the problem size so that each rank had approximately 1,000 atoms. On Cab, we

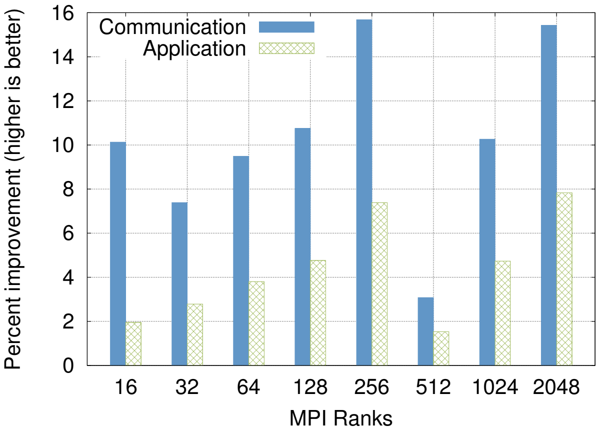

Figure 18: HMPI performance gains relative to MPI for MiniMD on the Cab system (16 ranks per node).

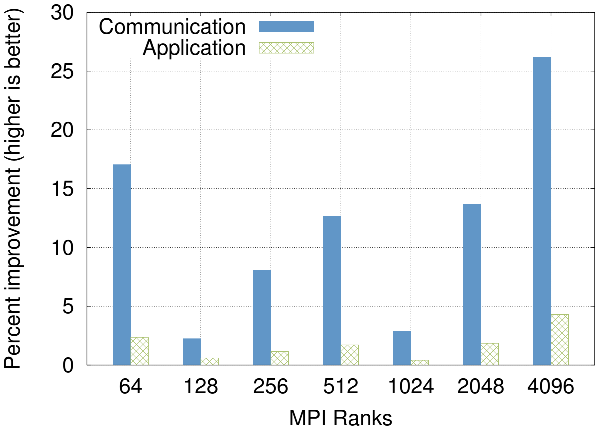

Figure 19: HMPI performance gains relative to MPI for MiniMD on the Sequoia system (64 ranks per node).

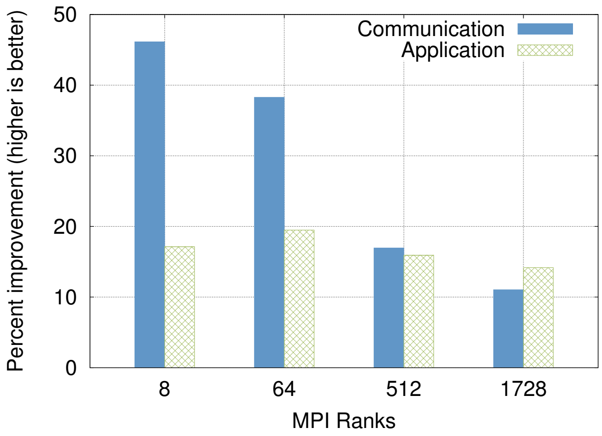

Figure 20: HMPI performance gains relative to MPI for Lulesh on the Cab system (16 ranks per node).

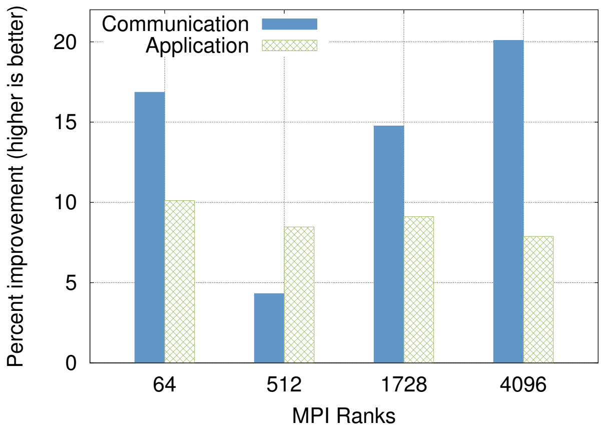

Figure 21: HMPI performance gains relative to MPI for Lulesh on the Sequoia system (64 ranks per node).

observe communication speedups ranging from 3.1-15.7%, resulting in total application time improvements of 1.5-7.9%. Sequoia shows improvements of 2.2-26.2% communication time and 0.2-4.1% application time. Compared to Cab, Sequoia has more memory and network bandwidth per FLOP. As a result, a smaller portion of execution time is spent communicating on Sequoia. Thus, changes in communication time have a smaller impact on overall application performance.

6.2 LULESH

LULESH, also known as Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics [10], is the mini-application version of a full-size hydrodynamics code in use at Lawrence Livermore National Laboratory. 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.

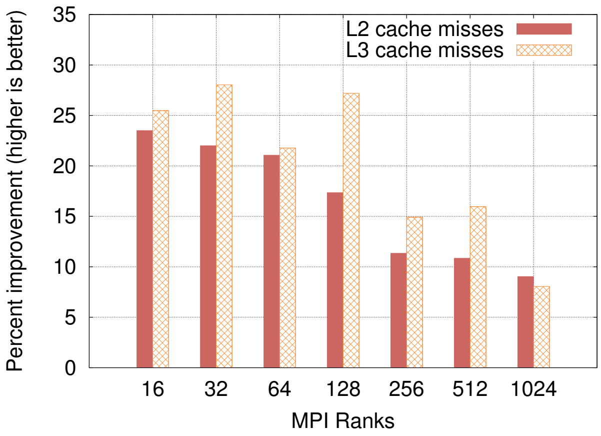

Figure 22: Reduction of total application cache misses in HMPI compared to MPI for MiniMD on Cab.

Figures 20 and 21 show 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 Cab, 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 Sequoia. As we saw with MiniMD, communication time speedups have a smaller impact on Sequoia compared to Cab.

6.3 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 and MVAPICH2. Figures 22 and 23 show results on the Cab system for MiniMD and LULESH, respectively. Cab'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. Cab has 16 cores per node, so only half of the available cores are utilized. More L3 cache is available per rank, resulting in fewer total misses for both HMPI and MPI.

7. CONCLUSION

In this paper we developed a novel scheme for exploiting shared memory hardware in MPI programs that are written for distributed memory systems. This mechanism shares *selected subsets* of memory among MPI ranks to implement communication between them more efficiently. However, the mechanism is not limited to MPI—it further enables optimizations such as ownership passing [2] or send/receive loop fusion, both departures from the one-copy semantics of MPI, or other application-specific optimizations.

In addition, we utilize a *layered* model for implementing our optimizations. HMPI resides on top of the MPI interface, however, it also integrates *vertically* in that it hooks into MPI's communications and transparently optimizes them. We showed that this model is highly portable and requires

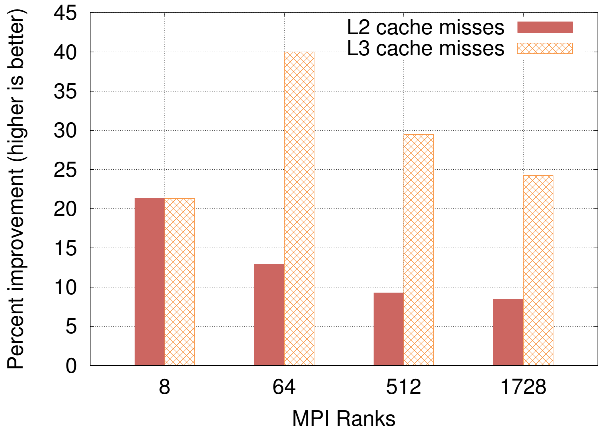

Figure 23: Reduction of total application cache misses in HMPI compared to MPI for LULESH on Cab.

no kernel extensions, system library changes, or administrative access. No modifications to applications or other MPI implementations are required.

We used these two mechanisms to develop HMPI, a fast layered MPI library that optimizes hybrid shared memory communication. Our optimizations show significantly less on-node communication overheads compared with traditional MPI approaches. Unlike prior work with thread-based MPI implementations, HMPI integrates transparently into legacy applications. We demonstrate the applicability with the MiniMD and LULESH application codes, which have communication time speedups of up to 26.2% and 46.1% respectively. Our shared memory communication techniques, implemented in HMPI, transparently improve performance for existing MPI applications on modern and future multi-core HPC systems.

8. ACKNOWLEDGMENTS

This work was supported in part by the Department of Energy X-Stack program and the Early Career award program. It was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

9. REFERENCES

[1] The OpenCL specification version 1.0, 2009.

[2] A. Friedley, T. Hoefler, G. Bronevetsky, C.-C. Ma, and A. Lumsdaine. Ownership passing: Efficient distributed memory programming on multi-core systems. February 2013. PPoPP 2013.

[3] E. Gabriel, M. Resch, and R. RÃijhle. Implementing mpi with optimized algorithms for metacomputing, 1999.

[4] B. Goglin and S. Moreaud. KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework. *Journal of Parallel and Distributed Computing*, 73(2):176–188, Feb. 2013. KNEM.

[5] M. A. Heroux, D. W. Dorfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. 2009.

[6] 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 *EuroMPI 2012, Vienna, Austria*, volume 7490, Sep. 2012.

[7] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, College Station, Texas, October 2003.

[8] G. Inc. gperftools. `https://code.google.com/p/gperftools`.

[9] R. Keller, E. Gabriel, B. Krammer, M. S. MÃijller, and M. M. Resch. Towards efficient execution of mpi applications on the grid: Porting and optimization issues. *Journal of Grid Computing*, 2003.

[10] Lawrence Livermore National Laboratory. Hydrodynamics challenge problem, 2012.

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

[12] S. Li, T. Hoefler, , and M. Snir. NUMA-Aware Shared Memory Collective Communication for MPI. Jun. 2013. HPDC'13.

[13] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *ACM International Conference on Supercomputing (ICS'03)*, 2003.

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

[15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9, 1991.

[16] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kale, and P. M. Ricker. Automatic MPI to AMPI Program Transformation using Photran. In *3rd Workshop on Productivity and Performance (PROPER 2010)*, number 10-14, Ischia/Naples/Italy, August 2010.

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

[18] K. Pedretti and B. Barrett. XPMEM: Cross-Process Memory Mapping.

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

[20] M. Pérache, P. Carribault, and H. Jourdren. MPC-MPI: An MPI implementation reducing the overall memory consumption. In *EuroPVM/MPI*, Berlin, Heidelberg, 2009.

[21] H. Tang and T. Yang. Optimizing threaded MPI execution on SMP clusters. In *ACM International Conference on Supercomputing (ICS)*, 2001.

[22] R. Thakur and W. Gropp. Test suite for evaluating performance of multithreaded MPI communication. *Parallel Comput.*, 35(12), Dec. 2009.

[23] D. Turner and X. Chen. Protocol-dependent message-passing performance on linux clusters. In

IEEE International Conference on Cluster Computing, CLUSTER '02, Washington, DC, USA, 2002.

[24] M. Woodacre, D. Robb, D. Roe, and K. Feind. The SGI Altix 3000 global shared-memory architecture. 2005.

[25] H. wook Jin, S. Sur, L. Chai, and D. K. Panda. Limic: Support for high-performance mpi intra-node communication on linux cluster. In *In International Conference on Parallel Processing (ICPP)*, pages 184–191, 2005.