Received 9 Oct 2017; | Revised 8 Nov 2017; | Accepted 8 Nov 2017
DOI: xxx/xxxx

EXAMPI 2017

Optimizing Point-to-Point Communication between Adaptive MPI Endpoints in Shared Memory

Sam White* | **Laxmikant V. Kale**

[1]Department of Computer Science, University of Illinois at Urbana-Champaign, IL, USA

Correspondence
*Sam White, Email: white67@illinois.edu

Abstract

Adaptive MPI is an implementation of the MPI standard that supports the virtualization of ranks as user-level threads, rather than OS processes. In this work, we optimize the communication performance of AMPI based on the locality of the endpoints communicating within a cluster of SMP nodes. We differentiate between point-to-point messages with both endpoints co-located on the same execution unit and point-to-point messages with both endpoints residing in the same process but not on the same execution unit. We demonstrate how the messaging semantics of Charm++ enable and hinder AMPI's implementation in different ways, and motivate extensions to Charm++ to address the limitations. Using the OSU microbenchmark suite, we show that our locality-aware design offers lower latency, higher bandwidth, and reduced memory footprint for applications.

KEYWORDS:
MPI, Shared Memory Optimizations, Intra-node communication, AMPI, Endpoints

1 | INTRODUCTION

Current High Performance Computing (HPC) systems already consist of thousands of nodes connected by a high-speed interconnect, with each node generally providing cache coherence across tens to hundreds of execution units. At the same time, the amount of memory available to each core is decreasing, and the total number of nodes seems unlikely to change as much as the node architecture itself. Because of this trend toward wider shared-memory nodes is clear, applications are now moving toward shared memory multithreading.

Along with applications, programming models and libraries are being adapted to this reality too. The Message Passing Interface (MPI) has not only made MPI+X safe, where X is a separate threading model, but has introduced shared-memory parallelism features of its own (1) (2). The MPI standard defines different threading levels to make the runtime aware of potentially unsafe resource sharing between application threads. Yet efficiently combining the two most commonly used models, MPI and OpenMP, remains a challenge for both runtimes and applications. The overheads incurred by MPI implementations in order to support MPI_THREAD_MULTIPLE have been well studied, with the alternative approach of serializing around communication having obvious drawbacks (3). Incremental improvements have been made to MPI implementations to improve hybrid performance (4) (5), but recent work has shown that restrictions to MPI's messaging matching semantics may be needed to scale MPI+X applications to current and future node architectures (6).

In this work we extend and optimize Adaptive MPI (AMPI) (7), a threaded implementation of the MPI standard, to take advantage of wider shared-memory nodes using the constructs exposed by its underlying runtime system, Charm++ (8). To this end, we examine AMPI's current performance on SMP clusters, optimize its message passing schemes to take advantage of user-space shared memory, and demonstrate the benefits of these optimizations using micro-benchmarks. In the context of this work, AMPI can be thought of as a restricted implementation of the MPI endpoints proposal (9) in which all endpoints are statically created as a part of MPI_COMM_WORLD. While some of our optimizations are particular to Charm++'s runtime system, other optimizations we employ are more broadly applicable to any threaded MPI implementation or any implementation of the MPI endpoints proposal.

2 | BACKGROUND

Adaptive MPI (AMPI) is an implementation of the MPI standard on top of Charm++ and its adaptive runtime system. AMPI currently supports the MPI-2.2 standard, and has support for most of MPI-3.1's features, such as non-blocking collectives, distributed graph virtual topologies, neighborhood collectives, large counts, and request-based RMA, with other features under development. AMPI lets users recompile preexisting MPI applications using its compiler wrappers and profit from its high-level runtime-automated features such as overdecomposition, communication/-computation overlap, dynamic load balancing, and automatic fault tolerance. AMPI has mostly been used by applications suffering from dynamic load imbalance which is often complicated to implement in a scalable fashion inside an application or library (10). AMPI's virtualization support has also been used to simulate application performance on various hardware system designs (11) (12).

In AMPI, the number of ranks in MPI_COMM_WORLD is a runtime parameter separate from the number of execution units. The ranks of MPI_COMM_WORLD are implemented as light-weight user-level threads, rather than as OS processes. User-Level Threads (ULTs) are used to allow fast context-switching between multiple ranks on an execution unit. The Charm++ runtime system schedules user-level threads in a non-preemptive manner based on the availability of messages for them. This leads to adaptive overlap of the communication of some ranks with the computation of others on the same execution unit. The Charm++ runtime system also provides mechanisms for migrating threads across the cores and nodes of the system. AMPI makes this migration of ranks transparent to users, who need only link with AMPI's Isomalloc memory allocator. Charm++'s support for measurement-based dynamic load balancing is available to AMPI users, as is the ability to perform coordinated migrations to storage (checkpoints) and to restart automatically from them without user or job scheduler intervention, possibly on a different number of cores.

The only drawback to the ranks-as-threads model is that it requires the application to not use global or static variables or to otherwise privatize them. Global and static variables are defined per OS process, and so are shared by all the AMPI ranks in a given process. Previous work has demonstrated how to manually convert existing codes and explored different automated approaches to solving the problem with compiler and runtime support (13).

Charm++ programs can be built and run in what is called 'SMP mode', in which one OS process is launched over multiple cores, with worker threads running the Charm++ scheduler on $n-1$ cores and a dedicated communication thread running on one core. Running in SMP mode provides some of the benefits of high-level, persistent multithreading to Charm++ applications, without requiring changes to the applications themselves. The difference between SMP and non-SMP mode is transparent to many Charm++ applications, though Charm++ provides some explicit interfaces to optimize for SMP mode and to safely call into non-thread-safe libraries. Charm++ applications often perform best at large scales in SMP mode and often run with one OS process either per node or per socket. Previously, AMPI has been able to run on the SMP mode of Charm++, but has not been explicitly optimized for it. In this work, we explain the shortcomings of Charm++'s messaging semantics for an MPI implementation and then optimize AMPI's point-to-point communication routines for SMP clusters. The rest of the paper is organized as follows: in section 3 we motivate the need for explicit changes to AMPI, in section 4, we profile the existing implementation of AMPI to guide our optmizations, in section 5 we describe a series of optimizations to AMPI, in section 6 we show micro-benchmark results for our new shared memory-aware design, and we conclude by summarizing our results and suggesting future extensions of this work.

2.1 | Related Work

Existing MPI implementations use different protocols to optimize for communication between ranks on the same hardware node. One method is to statically allocate intermediate buffers in shared memory, and to copy in and out the message payload data from these buffers. This method is used because registering shared memory pages across processes can be expensive compared to the cost of memory copy operations. Thus, MPI libraries reuse the same shared buffers for multiple operations. The copy-in/copy-out approach has also been optimized for High Bandwidth Memory and huge page sizes (14). For large messages, however, the cost of copying data becomes greater than that of dynamically registering and deregistering the memory, so most MPI implementations use kernel-assisted interprocess copy mechanisms to achieve 'zero copy' transfers, which copy directly from the user's send buffer to the user's receive buffer. SMARTMAP provided lightweight support for physical-address copies in the Catamount kernel (15), while other examples of kernel-assisted interprocess copy mechanisms include KNEM, CMA, LiMIC2, and XPMEM (16) (17) (18) (19). Our approach differs from these in that AMPI supports multiple virtual ranks in the same shared address space, and so can perform 'zero copy' transfers entirely in user-space.

Other MPI implementations have also included support for direct communication in shared memory. McMPI is an implementation of MPI developed natively in C# with such support (20). The PVAS model proposes to eliminate the process boundaries between communication endpoints (21). HMPI used a shared heap to accelerate transfers of data allocated on the heap, and included a parallel copy mechanism to further reduce large message latency (22). MPC-MPI is, similar to AMPI, a threads-based MPI implementation, and includes support for multiple ranks inhabiting the same address space (23) (24). AMPI, HMPI, and MPC-MPI all differ in their high-level features, and so may be targets for users facing different scaling problems. In contrast to the authors of HMPI, our optimizations also work for buffers on the stack. Our approach differs from the authors of MPC-MPI in that we optimize for the case where multiple ranks reside on the same execution unit, in our exploration of different optimizations for messaging in userspace shared memory, and in our focus primarily on message latency and bandwidth rather than memory footprint. Our work is also relevant to any implementation of the MPI endpoints proposal.

3 | ADAPTIVE MPI

3.1 | Overview

Adaptive MPI implements the MPI standard on top of Charm++. Charm++ is a C++-based object-oriented parallel programming system with an asynchronous many-task runtime system. Users are responsible for decomposing their application into C++ objects that can communicate via asynchronous remote method invocation and are scheduled in a non-preemptive, message-driven manner. Programmers are encouraged to over-decompose their problem into many more work and data units than there are execution units, so that the communication of some parallel objects can be overlapped naturally with computation of others on the same execution unit. Charm++ also provides infrastructure for migrating objects between address spaces at runtime, and for performing measurement-based dynamic load balancing as well as online fault tolerance.

Adaptive MPI virtualizes the ranks of MPI_COMM_WORLD so that users can run with more ranks than execution units. AMPI ranks are implemented as User-Level Threads (ULTs) associated with a Charm++ parallel object. ULTs are fast to context switch and can be co-scheduled by Charm++ alongside other parallel objects, allowing easy access to latency tolerance and overdecomposition in MPI applications. Charm++ also provides support for migrating ULT stacks and any heap memory associated with a ULT across address spaces at runtime for the purposes of load balancing and fault tolerance.

3.2 | Shared Memory Mode

AMPI, just like any other Charm++ application or library, can be built and run on Charm++'s SMP mode. In this mode, illustrated in figure 1, one operating system process is launched with n persistent threads on n execution units. We define an execution unit as a processing element that can have a thread affinitized to it, i.e. a core or a hyperthread. One thread is dedicated to handling communication, while the remaining $n-1$ threads run the Charm++ scheduler and have the application's parallel objects executed on them. For messages sent between objects inhabiting the same OS process, the runtime system passes the pointer to the message rather than copying the message or sending it through the Network Interface Card (NIC). The latency of messages passed in shared memory is then that of querying the locality of the recipient object, putting the message pointer in a concurrent queue on the recipient's execution unit, and the recipient dequeuing the message pointer and executing its task.

Even though AMPI already builds and runs on Charm++'s SMP mode, it fails to take advantage of the shared-memory semantics it provides. This is due to conflicting message buffer ownership semantics in MPI and Charm++. In MPI, message buffers are owned by the application. In Charm++,

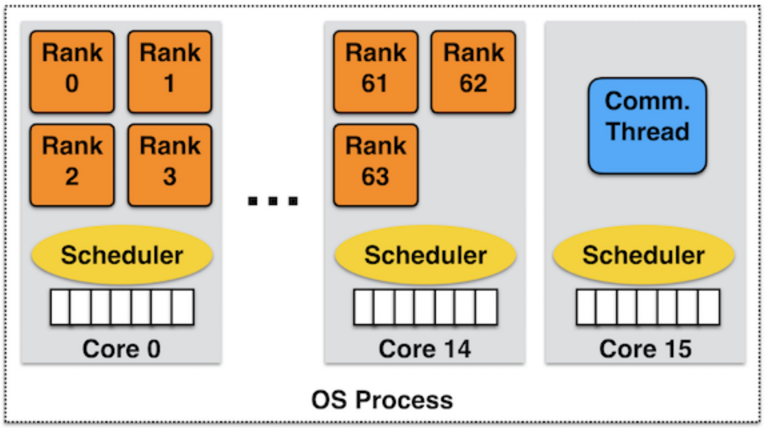

FIGURE 1 AMPI in SMP mode: this example shows 64 ranks running on a 16-core hardware node, with 1 core dedicated to handling communication. Users can also launch scheduler instances per hyperthread instead of per core. The number of ranks, the number of worker threads per process, and the number of processes are all command line arguments to the program.

messages are first-class objects, with their own metadata encapsulated. The runtime system assumes ownership of messages during a remote task invocation and provides ownership to the recipient object when the task is later executed. Consequently, Charm++ messages enable 'zero copy' messaging only if the application explicitly reuses the message objects in its own data structures. If not, users can pack and unpack data into messages or, more commonly, let Charm++ generate code to do the (de)serialization automatically via parameter marshaling. These semantics dictate that AMPI must always serialize from the sender's buffer into a message, and then deserialize from the message into the user-provided receive buffer on the other end, regardless of the datatype used.

In this paper, among other optimizations, we circumvent the extra copies needed in AMPI for communication within a shared-memory space. This includes communication using derived datatypes, and our method is completely portable without need for kernel-assisted interprocess copy and memory mapping mechanisms.

4 | PERFORMANCE ANALYSIS

4.1 | Experimental Setup

We run our experiments on Quartz, a Linux cluster at Lawrence Livermore National Laboratory composed of Intel Xeon E5-2695 ('Ivy Bridge') nodes, each with 36 cores running at 2.1 GHz (25). Cori is a Cray XC40 system at the National Energy Research Scientific Computing Center (26). We use its Haswell partition, which is made up of Intel Xeon E5-2698 v3 ('Haswell') nodes, each having 32 cores running at 2.3 GHz. We do not use hyperthreading on either system.

We use the OSU Micro-benchmarks suite, version 5.3.2 (27). We use the point-to-point latency and bidirectional bandwidth benchmarks, with the following modifications. We increase the maximum message size from 4 MB to 64 MB, and we change the request and status objects used in the non-blocking tests from global variables to local variables, and we combine the separate send and receive request arrays used in the bidirectional

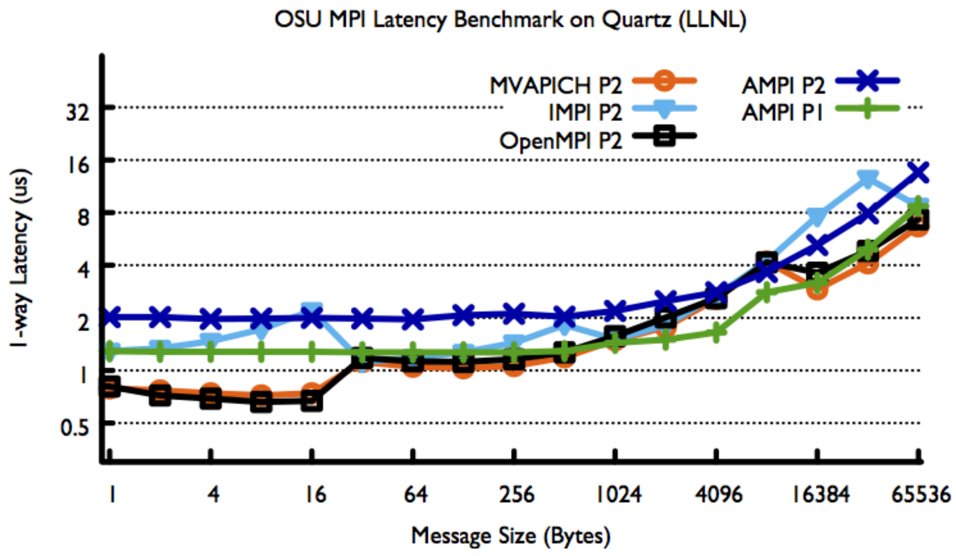

FIGURE 2 The existing AMPI implementation performs poorly in shared memory. MVAPICH2/2.2 and OpenMPI 2.0 perform the best.

bandwidth test to a single array in order to collapse the two consecutive calls to MPI_Waitall into a single call to MPI_Waitall. In general, placing two calls to Waitall back-to-back should be avoided in applications, since it artificially limits the amount of work that the MPI library can complete. All results shown are the average of 10 runs of the OSU benchmark specified.

On Quartz, we use the default Intel compiler, version 16.0.3. We show results for the default MVAPICH2/2.2 implementation as well as the pre-installed OpenMPI 2.0.0 and Intel MPI 2018.0 modules. On Cori, we use the default Intel compiler version 18.0.0.128 and compare against the default Cray MPI implementation version 6.7.0.

All results for AMPI are obtained using SMP mode to expose user-space shared memory between endpoints in the same process. We use 'AMPI P1' to mean both endpoints are co-located on a single execution unit and 'AMPI P2' to denote the case where two endpoints are running on two separate execution units in the same process.

4.2 | Existing Performance Issues

We distinguish between messages that are local to a given execution unit, meaning they travel between ranks co-located on the same execution unit, and messages that are local to a given process, meaning they travel between two ranks that reside on different execution units in the same address space. We maintain the distinction because the first case admits optimizations that the second does not.

Figure 2 shows the small message latency on 1 node of Quartz at LLNL for MVAPICH2/2.2, Intel MPI 2018, OpenMPI 2.0, and AMPI. For AMPI we separate the case where two ranks reside on the same execution unit (AMPI P1) and the case where they reside on different execution units in the same process (AMPI P2). We notice that despite having user-space shared memory available to it, AMPI does not perform well compared to the other MPI implementations.

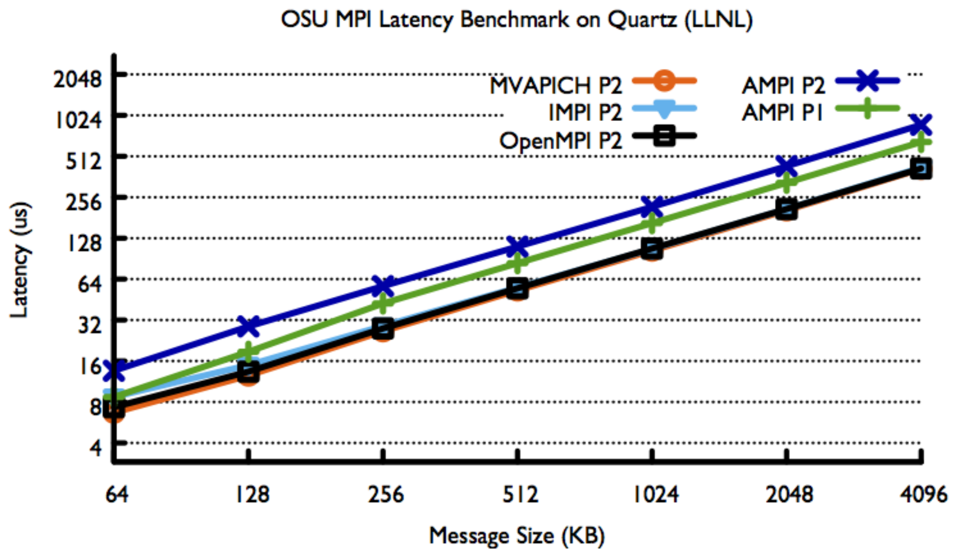

FIGURE 3 For messages sized 64KB to 4MB, all three process-based MPI implementation attain similar performance while AMPI P2 is consistently around 2x slower.

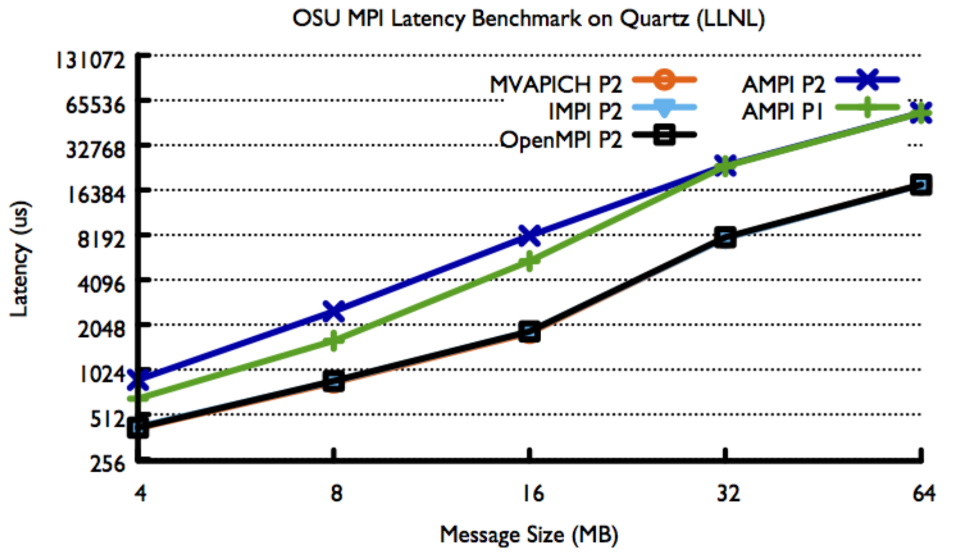

FIGURE 4 Large message latency suffers from an intermediate copy in AMPI P1 and AMPI P2.

For small messages, AMPI P2 consistently has the highest latency, with the exception being Intel MPI at a certain few sizes. AMPI is 2.81x worse than MVAPICH2/2.2 for 8 byte messages. AMPI P1 also has higher latency than MVAPICH2/2.2 and OpenMPI for many small message sizes. AMPI P1 is 81% slower than MVAPICH2/2.2 for 8 byte messages.

For large message sizes, seen in figures 3 and 4, AMPI's existing point to point communication performs even worse compared to the other MPI implementations. AMPI in both cases is 3x worse than the other MPI implementations for messages larger than 16MB. Looking at bandwidth utilization, AMPI fares no better, doing 2x worse at its peak bidirectional bandwidth usage than MVAPICH2/2.2, as shown in figure 5. In addition,

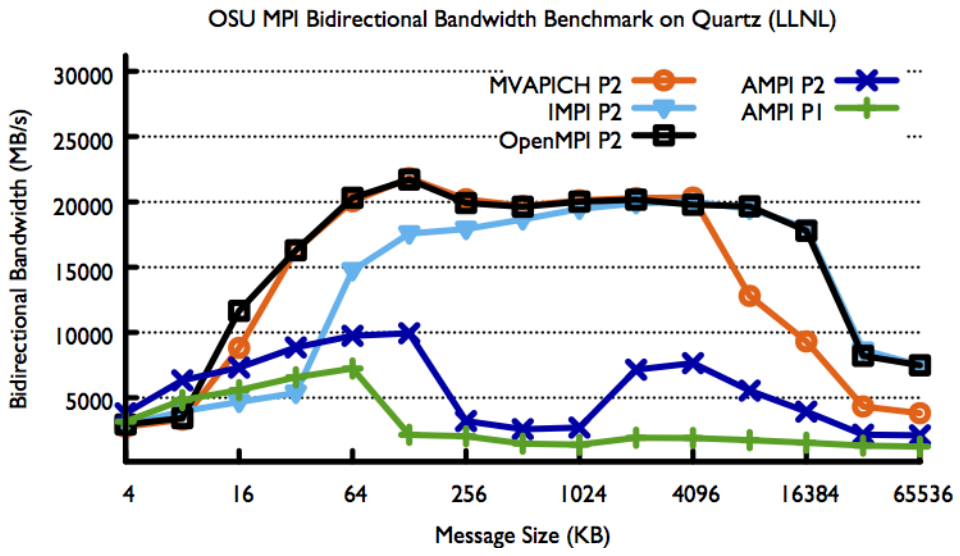

FIGURE 5 Bidirectional bandwidth results for messages sized 4KB to 64MB.

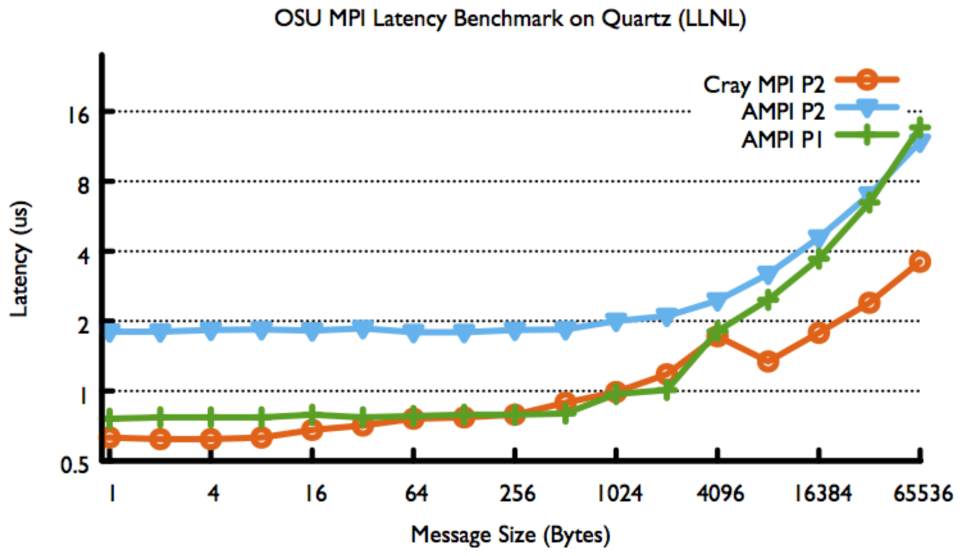

FIGURE 6 Latencies of small messages on the Haswell partition of Cori.

the same performance trends hold true on one node of the Haswell partition of Cori at NERSC. Figures 6 and 7 show that AMPI's small message latency and bidirectional bandwidth are both poor compared to Cray MPI.

5 | PERFORMANCE OPTIMIZATIONS

In order to understand why AMPI is performing so poorly, we profile its performance on the messaging latency benchmark. Table 1 breakdown the time spent in AMPI by three parts: time spent in Charm++ scheduling (this includes the ULT context switching time), time spent copying the message

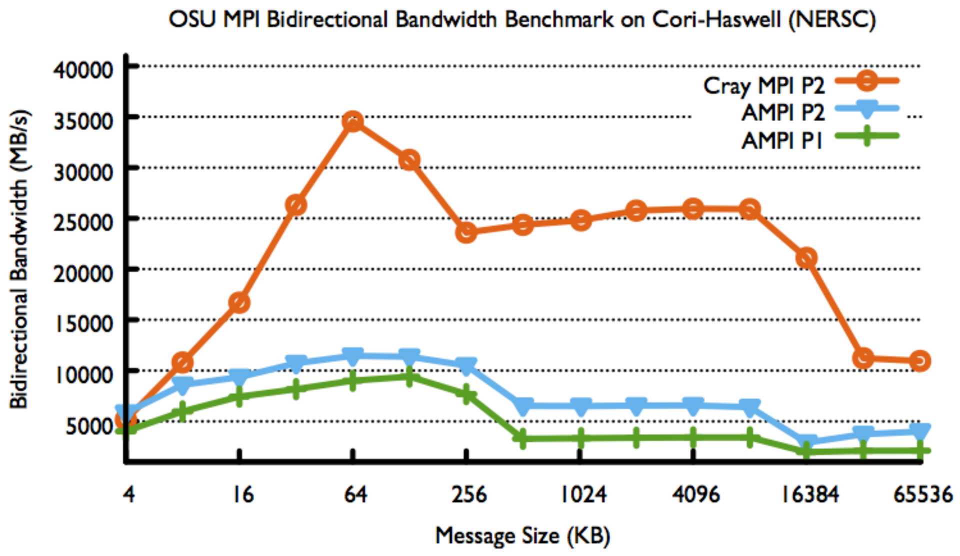

FIGURE 7 Bidirectional bandwidth of messages sized 4KB to 64MB.

Overhead per message	0-B message	1-MB message
Scheduling	1.02	1.04
Memory copy	0.00	162.86
Other	0.25	1.31

TABLE 1 Overhead per message in microseconds (μs). Breakdown of time spent inside AMPI per one-way message latency. Scheduling includes the ULT context switching overhead, memory copy is the time to copy the message payload, and other includes message matching and Charm++ message creation.

payload, and time spent otherwise inside AMPI (message creation and matching). We see that for small messages, much of the time is consumed within the scheduler, and that for large messages, an inordinate amount of time is spent in memory copy operations. The cost of message creation and matching increases slightly with the message size because the cache is polluted by the intermediate copy in the large message case.

5.1 | Scheduling Overheads & Optimizations

We optimize the scheduling overhead in three separate ways. First, we optimize the ULT context switching routines themselves. Charm++ already includes support for multiple implementations of its ULT interface. These include implementations based on pthreads, Windows fibers, Quick-Threads, the (deprecated) POSIX ucontext interfaces, and a jump-buffer based implementation. By default Charm++ uses the ucontext ULT implementation on most Linux-based platforms. However, replacing the ULT implementation with others, we found that QuickThreads performed the best. QuickThreads uses assembly instructions to save and restore only the registers that are needed for swapping ULT contexts. The time spent in scheduling and context switches went down from 1.02 μs to 415 ns when we switched from ucontext to QuickThread ULTs. Note that there are two context switches per eager message transmission: one from the sender thread to the scheduler, and another from the scheduler thread to the receiver thread.

Second, we reduce the number of context switches needed when executing MPI_Waitall, which is used in the bidirectional bandwidth benchmark. Previously, AMPI would set a counter in MPI_Waitall equal to the number of requests, then would test all requests for completion and decrement the counter for each completed request. If after looping over the array of requests there remained one or more incomplete requests, the thread would suspend itself until a message arrived. Once any message for that rank arrived, the thread would be awoken and would repeat the check of all requests. This loop would execute until all requests had completed. Consequently, messages that matched requests that were not currently being blocked on would resume the thread only to accomplish no effective progress, and messages that did match a blocked on request would result in a context switch even if that request was not the last one needed to complete the Waitall operation.

In order to avoid unnecessary context switches inside MPI_Waitall, we associate a counter with each rank of the number of requests it is currently blocked on, and we add a boolean flag to each request object that specifies if it is currently blocked on or not. Inside Waitall, we now check all requests for completion once, marking incomplete ones as ‘blocked on’ and incrementing the rank’s counter. Then if the counter is nonzero, the thread remains suspended. As messages arrive, if they match a request that is currently blocked on, we decrement the rank’s counter. If after processing a matched message, the rank’s counter is zero, that rank’s thread is awoken. This ensures that the thread is only awoken once it can complete the entire Waitall operation. The benefit of maintaining the counter as part of the rank’s state is visible in the bidirectional bandwidth benchmark for small messages, which improves 2.17x for 64 byte messages from 100.62 MB/s to 217.05 MB/s.

Third, we observe that AMPI messages incur unnecessary trips through the scheduler. In AMPI the sender would copy its buffer into a Charm++ message, stick the message in the Charm++ scheduler queue, and eventually suspend itself upon reaching a blocking MPI call. Then, the message would be delivered to a task on the receiver, who would potentially match the message to a request object, deserialize the messsage’s payload to the receiver’s buffer, and potentially resume the receiver’s thread. Instead of taking this trip through the scheduler, we can look up the receiver’s local AMPI object and call methods on it directly as a C++ object. Charm++ has support for making this local object lookup automatic in what it calls ‘inline’ entry methods (tasks), which execute inline if the callee object is on the same execution unit as the caller, and otherwise sends a message. Using inline entry methods reduces the number of trips through the Charm++ scheduler, lowering the latency of all local communication calls. With these optimizations in place, the latency of small messages is reduced from 1.29 μs to 0.73 μs on one execution unit of Quartz.

5.2 | Memory Pooling

The majority of the cost is still inside the ULT context switching, which cannot be avoided entirely. Of the remaining costs for small messages, we noticed that almost all of the time was spent in four data structures inside AMPI: message creation/deletion, request creation/deletion, and the two message matching queues for posted requests and unexpected messages. For its message matching queues, AMPI was dynamically creating and deleting queue entry objects for each insertion or removal. For all of these data structures, we replaced dynamic memory allocation with memory pools. Request objects and matching queue entries are fixed-size objects, but messages can have arbitrary sizes. We set a threshold size below which we first check the message pool for one that has been pre-allocated. We set the pooled message size threshold to be by default greater than the size of the short messages used in first step of the rendezvous protocol (64 bytes) described in the next section. By maintaining memory pools of these objects, we eliminate all dynamic memory allocation in the fast path of AMPI's point-to-point messaging protocols. This brings the latency of all messages 64 bytes or smaller down further from 0.73 μs to 0.61 μs for messages sent and received on the same execution unit, and from 1.14 μs to 0.97 μs for messages between execution units in the same address space.

5.3 | Large Message Optimizations

Next we look at latencies for large messages in AMPI and identify opportunities for performance improvement. While the 'inline' entry method optimization effectively reduces the scheduling overhead, AMPI still pays the price of copying the message payload twice: first to copy or serialize the buffer into a Charm++ message object on the sender and second to copy or deserialize from the message object to the user's buffer on the receiver.

5.3.1 | Locality within an Execution Unit

For messages sent between endpoints co-located on the same execution unit, no synchronization or locking is required around accesses to another rank's internal messaging data structures. Since AMPI restricts users to MPI_THREAD_FUNNELED, we know that only a thread spawned by AMPI can ever call into the runtime. Consequently, if a given rank is running, we know that none of the other ranks on its execution unit can be active inside the runtime, and so no synchronization is needed around accesses to other rank's internal data structures on the same execution unit. The sender can directly peek into the receiver object's data structures to determine if its message's matching request is preposted. Avoiding the intermediate copy of the message payload is easy when the message is expected and the receiver resides on the same execution unit. In this case a single copy operation can be performed from the sender's buffer to the receiver's buffer.

For the case that the message is unexpected, we implemented a rendezvous protocol in AMPI to avoid making an intermediate copy. Thus, when a message is being sent on the same execution unit the sender first checks if its message is expected or not. If not, it deposits a short message in the receiver's unexpected message queue which the receiver will later match. This message contains an object that stores the sender's buffer address, count, process number, a pointer to the sender's datatype object, and a callback object. This object can be used by AMPI when the receiver subsequently matches the message to determine if the sender still resides in the same shared address space, how to perform the copy, and how to notify the sender when it is done. The callback specifies the MPI_Request corresponding to the send request allocated on the sender, so that when the callback is invoked the sender can lookup and complete the corresponding request object. If the receiver migrates out of the process that the sender thought it was in when it sent its short message, the receiver will send a request back to the sender to effectively resend the data over the network. This case is slower, but should happen only rarely, i.e. in the first iteration after a call to a load balancer resulting in migrations across processes.

5.3.2 | Locality within a Process

All of the optimizations described so far apply to messages between endpoints residing on different execution units as well as the single execution unit case. The main difference between the two scenarios is that across execution units we cannot assume that it is safe to access another endpoint's internal messaging state. In order to synchronize concurrent accesses to the messaging data structures, we rely on Charm++'s message-driven scheduling. Alternative designs would associate one big lock with each endpoint's internal state or finer-grain locking throughout AMPI's data structures. Such designs would need to maintain a table of pointers to all ranks in a given process, and when messaging another rank would need to acquire safe access to its data structures. Instead, we use Charm++ messages for all synchronization between execution units in AMPI. The rendezvous protocol is essentially the same as it was described earlier for one execution unit, with a couple exceptions: one, the short message with the object describing the send buffer is pushed into Charm++ queue on the receiver's execution unit. Sometime later that message is scheduled and the receiver performs the direct user-space memory copy operation. Since the message goes through the scheduler rather than happening inline, the sender creates a send request to track completion of the send buffer. When the receiver is finished with the copy, it sends a message back to the sender so that it can complete its send request. Finally, the receiver thread awakened if the request just completed was being blocked on. Compared to the eager method, this rendezvous protocol adds the cost of completing the sender's request to the total latency. However, it does so to trade-off time spent in memory copy operations on the sender. We find that on Quartz, the cross-over point between the eager and rendezvous protocols is 4 KB in terms of latency.

6 | RESULTS

We call our shared memory-aware implementation 'AMPI-shm'. Figure 8 shows that on Quartz AMPI-shm has 2x lower latency than the previous AMPI implementation, and is now within 33% of MVAPICH2/2.2 for all message sizes under 1024 bytes. AMPI-shm provides lower latency than MVAPICH2/2.2 for all message sizes greater than 1024 bytes and is up to 2.33x faster for 64MB messages (figures 9-10). For messages between endpoints co-located on the same execution unit, AMPI-shm has 58% lower latency than AMPI for small messages.

In addition to providing lower latency, the node-aware implementation can use up to 2.76x higher bidirectional bandwidth than before. Compared to MVAPICH2/2.2, it has 26% higher bidirectional bandwidth for large messages (figure 11). AMPI-shm P1 also provided 2.31x higher bidirectional bandwidth than AMPI P1. In fact, AMPI can now saturate all of the main memory bandwidth that is available to two cores of a node for messaging. We measured the maximum memory bandwidth using the STREAM copy benchmark (28), which on two cores of Quartz achieved a bandwidth of 25,926 MB/s. No process-based MPI implementation on Quartz was able to reach the memory bandwidth limit.

On Cori, we see similar improvements using the shared memory-aware AMPI implementation compared to the previous implementation. For the small messages sizes shown in figure 12, AMPI-shm has 2x lower latency than AMPI for both cases of co-located endpoints and endpoints on different execution units in the same process. For 0 byte messages, our new implementation has an average latency of only 380 ns. For large messages, seen in figures 13 and 14, AMPI-shm provides up to 3.99x lower latency. For 0 byte messages, Cray MPI performs better, while for 64

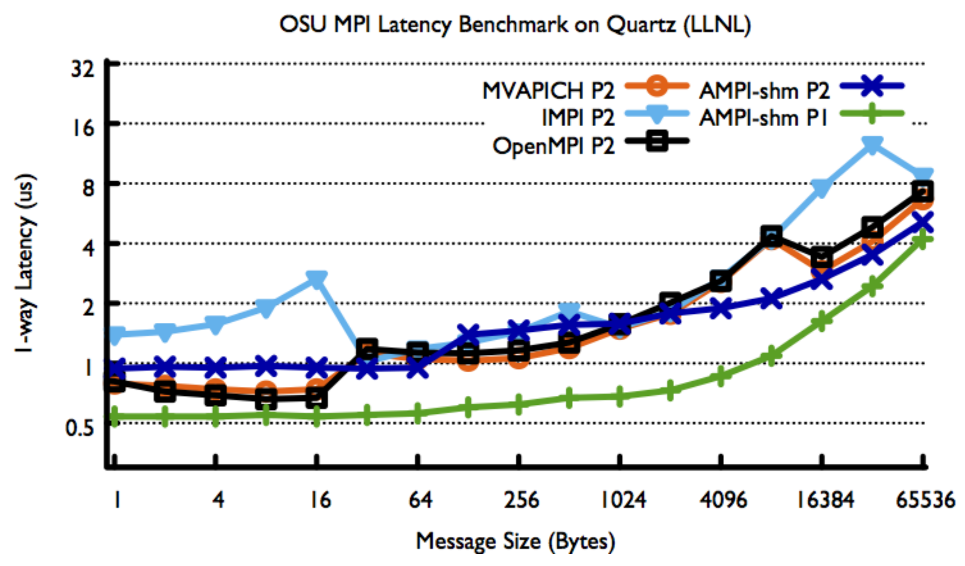

FIGURE 8 For small messages, our design brings AMPI within 33% of the best process-based MPI implementation.

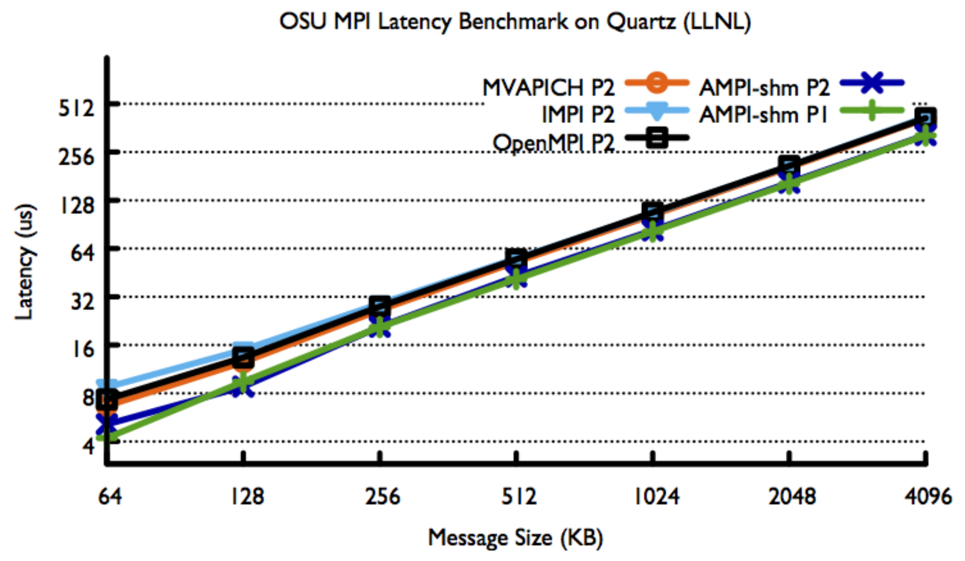

FIGURE 9 For large messages, the user-space single copy method achieves lowest latency.

MB message sizes, AMPI-shm is 37% faster. For nearly all message sizes greater than 4 KB and less than 32 MB, Cray MPI and AMPI-shm perform nearly identically in terms of latency. We expect Cray MPI's performance is due to the lower overhead of the XPMEM kernel module on Cray's operating system compared to the kernel modules and interprocess copy mechanisms (CMA, LiMIC2) used by MPI implementations on commodity Linux clusters such as Quartz.

For the bidirectional bandwidth benchmark, shown in figure 15, we see AMPI-shm and Cray MPI both achieve full bandwidth utilization. On Cori, STREAM copy gives a maximum memory bandwidth of 33,760.24 MB/s on two cores.

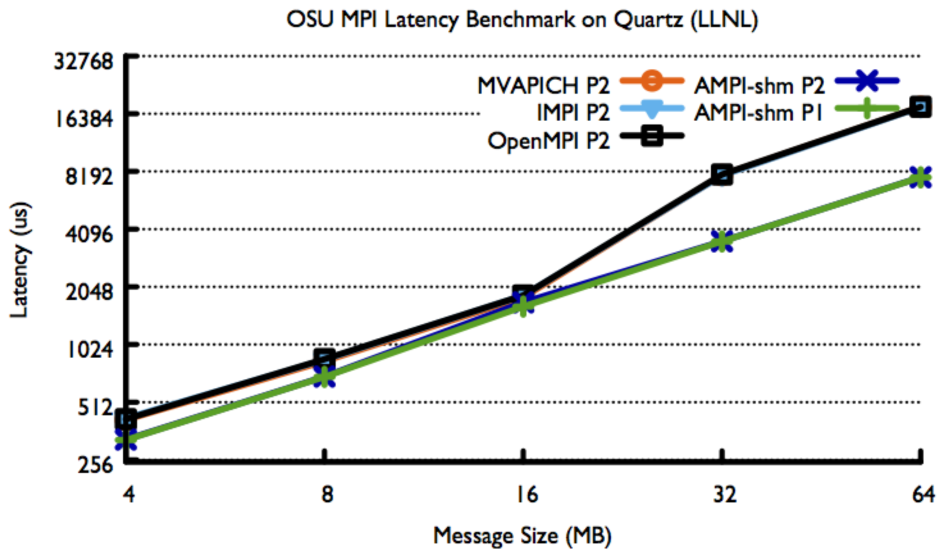

FIGURE 10 At the largest message sizes of 32 and 64 MB, AMPI-shm P1 and P2 have 2.33x better latency than the process-based MPI implementations.

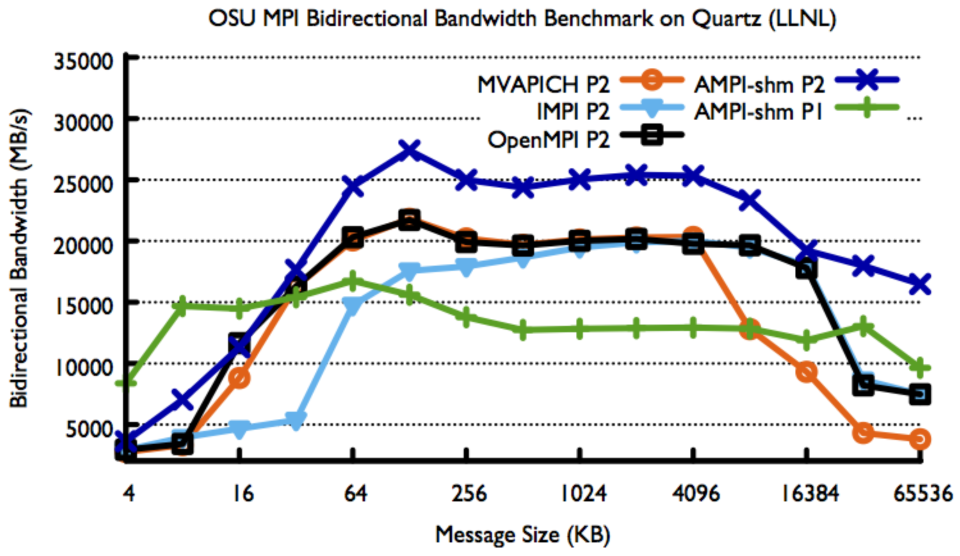

FIGURE 11 The shared memory-aware implementation of AMPI outperforms process-based MPI implementations in terms of bidirectional bandwidth. STREAM copy achieves a bandwidth of 25,926 MB/s on two cores of Quartz.

7 | EXTENSIONS & FUTURE WORK

7.1 | Collectives

AMPI currently implements most of MPI's collective routines by using Charm++'s equivalent collective operations. Charm++ provides built-in support for all kinds of reduction operations, allreduce, barriers, broadcasts, gathervs, and allgathervs. Since AMPI uses these routines directly, its collective implementations do not benefit transparently from improvements to its point-to-point communication routines. In the future we plan to

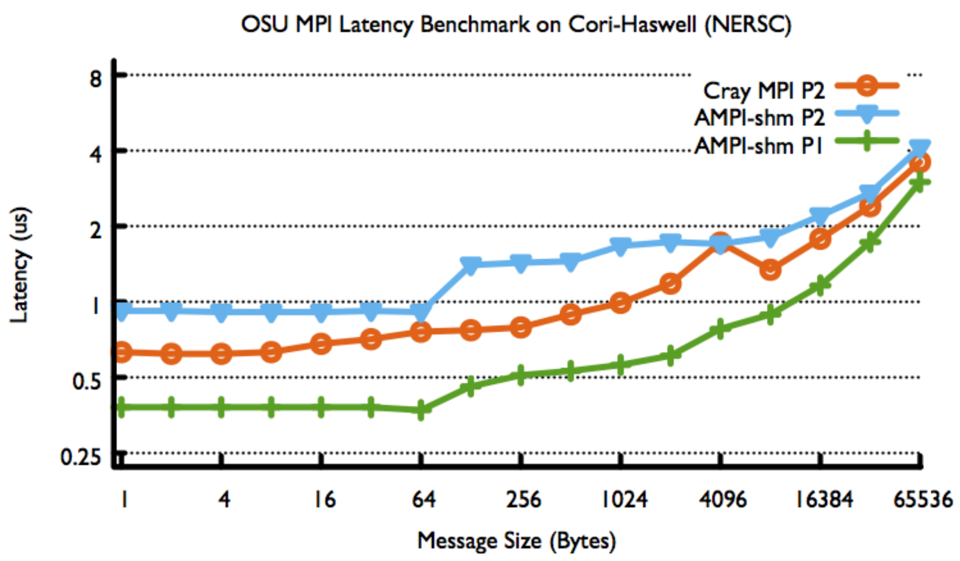

FIGURE 12 For small messages on a single execution unit, AMPI-shm outperforms Cray MPI. For messages between execution units, Cray MPI is still 1.44x faster.

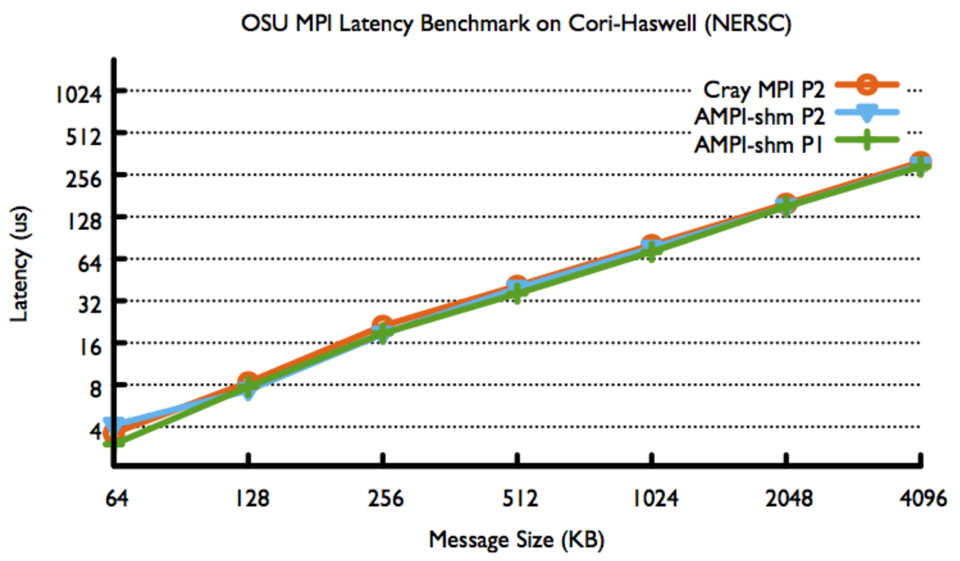

FIGURE 13 For messages sized 64KB to 4MB, AMPI-shm performs similarly to Cray MPI.

add support for hierarchical collectives, in which all ranks in the same shared address space can aggregate their contributions in shared memory before or after off-node communication is done as necessary.

7.2 | Derived datatypes

A naive design supporting derived datatypes would send a short message over for each contiguous part of the sender's non-contiguous buffer, resulting in as many Charm++ messages being created and scheduled as there are contiguous parts of the non-contiguous datatype. We currently

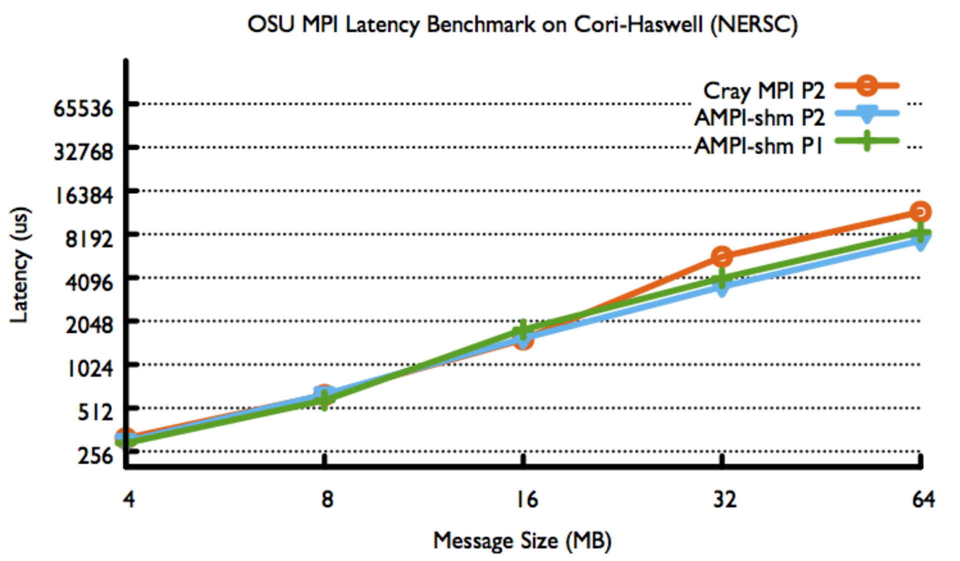

FIGURE 14 At the largest message sizes of 32 and 64 MB, AMPI-shm P1 and P2 have 37% lower latency than Cray MPI.

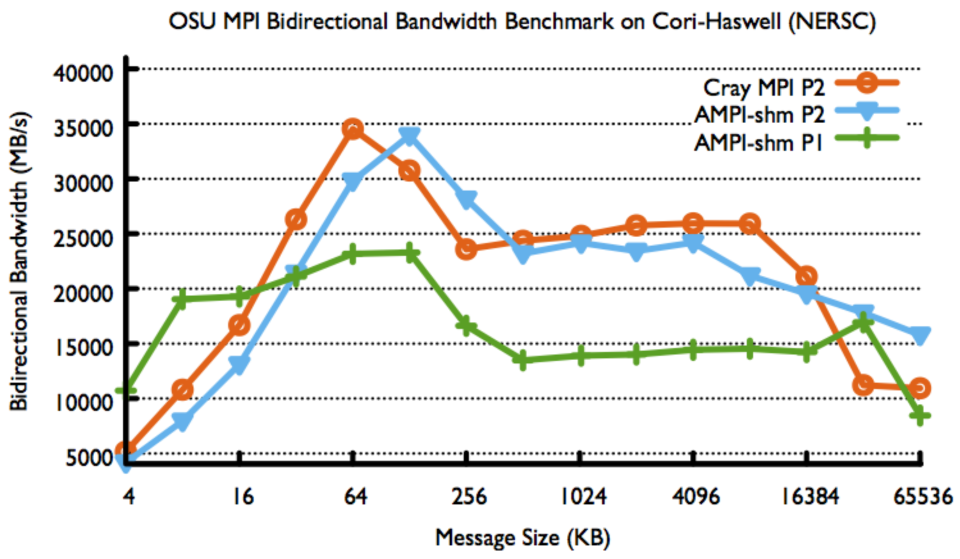

FIGURE 15 AMPI-shm P2 performs similarly to Cray MPI in terms of bidirectional bandwidth. STREAM copy achieves a bandwidth of 33,760.24 MB/s on two cores of Cori's haswell partition.

send a pointer to the sender's internal datatype object in the short message so that the receiver can use it to do the direct copy. This in principle allows for a direct copy from a buffer with a non-contiguous send datatype to a buffer with a non-contiguous receive datatype; however, AMPI's internal datatypes library does not yet support making direct copies between two non-contiguous datatypes, so an intermediate buffer is created and used.

7.3 | Shared memory windows

AMPI does not yet support MPI-3's shared memory windows, but we intend to implement that interface with optimizations for user-space shared memory in the future. Kernel-assisted shared memory interfaces can incur TLB overheads when accessing pages from other processes, as we have seen in this work.

7.4 | 'Zero copy' communication across processes

We note that kernel-assisted interprocess copy mechanisms could be used to extend this work for the case where multiple processes are launched within a single hardware node. This is usually done on nodes with multiple sockets or NUMA architectures, typically launching one process per NUMA domain.

In order to support 'zero copy' transfers across processes in AMPI, the semantics of Charm++ messaging must be extended. We demonstrated how to perform direct userspace copies across execution units in a single Charm++ process, but to extend our approach to messages across nodes we would need to use the underlying communication APIs for Remote Direct Memory Access (RDMA). Charm++ internally uses RDMA already on its own message objects if they are above certain threshold sizes, but again the semantics of a first class message object do not match well with MPI's. If Charm++ exposed an API for performing remote put and get operations, AMPI's rendezvous protocol would only need to be extended to handle the case where the copy operation happens asynchronously rather than inline, and memory pinning and unpinning would need to be done on networks that require it. The asynchronous completion aspect could be implemented by adding a callback on the receiver to look up and complete the request object. Such an API would also benefit Charm++ libraries, allowing them to communicate data in-place without requiring changes to the application's data structures.

8 | CONCLUSIONS

We presented a shared memory-aware implementation of Adaptive MPI that optimizes for user-space shared memory between endpoints inhabiting the same process. We further optimized for the case where multiple endpoints are co-located on the same execution unit. Our results show that our new design, AMPI-shm, performs much better in terms of both latency and bandwidth compared to the previous AMPI implementation. On Quartz, AMPI-shm has 2x lower latency than the previous AMPI implementation and provides lower latency than MVAPICH2/2.2 for all message sizes greater than 1024 bytes. At message sizes of 64 MB, it is up to 2.33x faster than all process-based MPI implementations on Quartz. For messages between endpoints co-located on the same execution unit, AMPI-shm has 58% lower latency than AMPI for small messages.

In addition to providing lower latency, the node-aware implementation can use up to 2.76x higher bidirectional bandwidth than before, and now can saturate all of the main memory bandwidth for messaging on both systems we tested. For messages of 64 MB, our implementation achieves 3.99x higher bandwidth than MVAPICH2/2.2 on Quartz. Since AMPI does its messaging using a single direct copy in user-space, there is no need for intermediate buffers or for non-portable kernel-assisted interprocess copy mechanisms. Overall, our shared memory-aware messaging protocols provide improved latency and bandwidth compared to the previous AMPI implementation and compared to process-based MPI implementations.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002374.

References

[1] Hoefler Torsten, Dinan James, Buntinas Darius, et al. Leveraging MPI's One-sided Communication Interface for Shared-memory Programming. In: EuroMPI'12:132–141Springer-Verlag; 2012; Berlin, Heidelberg.

[2] Hoefler Torsten, Dinan James, Buntinas Darius, et al. MPI+MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Computing*. 2013;95(12):1121–1136.

[3] Rabenseifner Rolf, Hager Georg, Jost Gabriele. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: PDP '09:427–436IEEE Computer Society; 2009; Washington, DC, USA.

[4] Amer Abdelhalim, Lu Huiwei, Wei Yanjie, Balaji Pavan, Matsuoka Satoshi. MPI+Threads: Runtime Contention and Remedies. *SIGPLAN Not.*. 2015;50(8):239–248.

[5] Dang Hoang-Vu, Seo Sangmin, Amer Abdelhalim, Balaji Pavan. Advanced Thread Synchronization for Multithreaded MPI Implementations. In: CCGrid '17:314–324IEEE Press; 2017; Piscataway, NJ, USA.

[6] Dang Hoang-Vu, Snir Marc, Gropp William. Towards Millions of Communicating Threads. In: EuroMPI 2016:1–14ACM; 2016; New York, NY, USA.

[7] Huang Chao, Lawlor Orion, Kalé L. V.. Adaptive MPI. In: :306-322; 2003; College Station, Texas.

[8] Acun Bilge, Gupta Abhishek, Jain Nikhil, et al. Parallel Programming with Migratable Objects: Charm++ in Practice. In: SC; 2014.

[9] Dinan James, Grant Ryan E, Balaji Pavan, et al. Enabling Communication Concurrency Through Flexible MPI Endpoints. *Int. J. High Perform. Comput. Appl.*. 2014;28(4):390–405.

[10] Huang Chao, Zheng Gengbin, Kumar Sameer, Kalé Laxmikant V.. Performance Evaluation of Adaptive MPI. In: ; 2006.

[11] Jain Nikhil, Bhatele Abhinav, White Sam, Gamblin Todd, Kale Laxmikant V.. Evaluating HPC Networks via Simulation of Parallel Workloads. In: SC '16 (to appear); 2016.

[12] Zheng Gengbin, Kakulapati Gunavardhan, Kalé Laxmikant V.. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In: :78; 2004; Santa Fe, New Mexico.

[13] Besnard Jean-Baptiste, Adam Julien, Shende Sameer, et al. Introducing Task-Containers As an Alternative to Runtime-Stacking. In: EuroMPI 2016:51–63ACM; 2016; New York, NY, USA.

[14] Cho Joong-Yeon, Jin Hyun-Wook, Nam Dukyun. Enhanced Memory Management for Scalable MPI Intra-node Communication on Many-core Processor. In: EuroMPI '17:10:1–10:9ACM; 2017; New York, NY, USA.

[15] Brightwell Ron, Pedretti Kevin, Hudson Trammell. SMARTMAP: Operating System Support for Efficient Data Sharing Among Processes on a Multi-core Processor. In: SC '08:25:1–25:12IEEE Press; 2008; Piscataway, NJ, USA.

[16] Moreaud S., Goglin B., Namyst R., Goodell D.. Optimizing MPI communication within large multicore nodes with kernel assistance. In: :1-7; 2010.

[17] Chai L., Hartono A., Panda D. K.. Designing High Performance and Scalable MPI Intra-node Communication Support for Clusters. In: :1-10; 2006.

[18] Vienne Jerome. Benefits of Cross Memory Attach for MPI Libraries on HPC Clusters. In: XSEDE '14:33:1–33:6ACM; 2014; New York, NY, USA.

[19] Jin Hyun-Wook, Panda Dhabaleswar K.. LiMIC: Support for High-Performance MPI Intra-node Communication on Linux Cluster. In: ICPP '05:184–191IEEE Computer Society; 2005; Washington, DC, USA.

[20] Holmes Daniel, Booth Stephen. McMPI: A Managed-code MPI Library in Pure C#. In: EuroMPI '13:25–30ACM; 2013; New York, NY, USA.

[21] Shimada Akio, Hori Atsushi, Ishikawa Yutaka. Eliminating Costs for Crossing Process Boundary from MPI Intra-node Communication. In: EuroMPI/ASIA '14:119:119–119:120ACM; 2014; New York, NY, USA.

[22] Friedley Andrew, Bronevetsky Greg, Hoefler Torsten, Lumsdaine Andrew. Hybrid MPI: efficient message passing for multi-core systems. In: :18ACM; 2013.

[23] Perache Marc, Carribault Patrick, Jourdren Herve. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In: Ropo Matti, Westerholm Jan, Dongarra Jack, eds. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proceedings of the 16th European PVM/MPI UsersâĂŹ Group Meeting (EuroPVM/MPI 2009)*, Lecture Notes in Computer Science, vol. 5759: Springer Berlin Heidelberg 2009 (pp. 94-103).

[24] Perache Marc, Jourdren Herve, Namyst Raymond. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In: Euro-Par âĂŹ08:78âĂŞ88Springer-Verlag; 2008; Berlin, Heidelberg.

[25] Quartz - Top500 List https://www.top500.org/system/178971Accessed: 2017-11-13; .

[26] Cori - Top500 List https://www.top500.org/system/178924Accessed: 2017-11-13; .

[27] OSU Micro-benchmarks suite http://mvapich.cse.ohio-state.edu/benchmarks/Accessed: 2017-9-10; .

[28] McCalpin John D.. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter.* 1995;:19–25.

☐